東京大学
情報理工学系研究科 創造情報学専攻
修士論文

動的コンパイルのオフロードが実現する対話的で省メモリなプログラミング環境の研究

A study of interactive and memory-saving programming environment with offloaded dynamic compilation.

前島 文香
Fumika Maejima

指導教員　千葉 滋 教授

2024 年 1 月

概要

近年 IoT への注目の高まりなどから，マイクロコントローラ向けのアプリケーション開発の需要は益々高まっている．マイクロコントローラの開発において開発環境が対話的であれば細い調整の度に，時間のかかるコンパイルやフラッシュへの書き込みを行わずに済むため便利である．しかしながら，既存の対話的な開発環境では受け取った文字列の処理から実行までをマイクロコントローラの限られたリソース内で行っているため，プログラムの実行速度が遅い．特に対話的な環境を高速化するための機能である動的コンパイラは多くのメモリを必要とするため，マイクロコントローラのメモリを圧迫する．そこで本研究では対話的で高速なマイクロコントローラ向けの言語処理系を実現するために，動的コンパイラのオフロードを提案する．マイクロコントローラよりも性能が高くメモリの豊富なホストマシン上で動的コンパイルを行えば，マイクロコントローラのメモリを圧迫することなく高速化できる．本研究では提案手法を BlueScript の処理系として実装した．BlueScript は対話的な開発環境を持った TypeScript 風の言語である．性能評価として，BlueScript と MicroPython，C 言語を用いて実行速度とプログラムの変更にかかる時間，ランタイムのバイナリサイズを比較した．

Abstract

With the increasing attention to IoT in recent years, the demand for application development for microcontrollers is increasing. An interactive development environment is useful for developing applications for microcontrollers because it eliminates the need for time-consuming compilations and writes to flash every time a small change is made. However, existing interactive development environments have a slow program execution speed because the process from processing received character strings to execution is performed within the limited resources of the microcontroller. In particular, dynamic compilers, a feature designed to speed up interactive environments, require a lot of memory, which puts pressure on the microcontroller's memory. In this paper, we propose a new archtecture of language processor with offloaded dynamic compiler to host machine, which allows us to archive both interactivity and high program execution speed. By performing dynamic compilation on a host machine that has higher performance and more memory than the microcontroller, we can speed up the process without putting pressure on the microcontroller's memory. We implemented the proposed method as a BlueScript processing system. BlueScript is a TypeScript-like language with an interactive development environment. For performance evaluation, we compared execution speed, time required to change the program, and runtime binary size using BlueScript, MicroPython, and C language.

目次

第 1 章　はじめに　1

第 2 章　対話的な開発環境とプログラムの実行速度　3
2.1　マイクロコントローラ　3
2.2　マイクロコントローラの開発における対話的な開発環境の必要性　5
2.3　対話的な開発環境と実行速度　7
2.4　動的コンパイラとメモリ使用量　10
2.5　関連研究　12

第 3 章　提案：動的コンパイラをホストマシンにオフロードする．　13
3.1　動的コンパイラのオフロード　13
3.2　BlueScript とその開発環境　14
3.3　動的コンパイラの設計と実装　15
3.4　BlueScript の言語仕様と実装　22

第 4 章　実験　32
4.1　実験環境　32
4.2　対話性　32
4.3　実行速度　40
4.4　バイナリサイズ　43

第 5 章　まとめと今後の課題　44

発表文献と研究活動　45

参考文献　46

第 1 章

はじめに

マイクロコントローラの開発環境が対話的であれば，効率的に開発を行うことができる．マイクロコントローラの開発は実機を用いた細かい調整が必要になることが多い．例えばマイコンカーの速度や，デバイスのディスプレイ上のどの座標に文字を表示するのかは通常，実機で動かしながら微調整を行う．もしこのようなアプリケーションの開発を C 言語のようにコンパイルとフラッシュへの書き込みを行わないと実行できない開発環境で行うならば，試行錯誤のたびに決して短くない待ち時間が生じる．例えば ESP32 向けの C 言語のコンパイラツールである ESP-IDF では，微調整をするたびにコンパイルと書き込みのために 7–8 秒程度の待ち時間が生じる．もし開発環境が対話的であれば，このような待ち時間は生じないだろう．

このような利点がありながら，マイクロコントローラの開発において対話的な環境は広く使われていない．その理由は，対話的なインターフェースを提供する開発環境ではプログラムの実行速度が遅いためである．従来のマイクロコントローラ上の対話的な開発環境は，マイクロコントローラ上に処理系を搭載し，ホストマシンから文字列を受け取って処理する方法で実装されていた．この方法では，マイクロコントローラのメモリが限られているため十分な最適化が行えず実行速度が遅くなっていた．特に，対話的な言語処理系を高速化するための典型的な技術である動的コンパイラは処理系の大きさを大幅に増やすため搭載することができなかった．

そこで，我々は動的コンパイラを開発に使用中のホストマシンにオフロードする方法を提案する．対話的に追加されたプログラム片をそのままマイクロコントローラに送るのではなく，ホストマシン上でコンパイルと最適化を行い実行可能なバイナリに変換してマイクロコントローラに送信する．コンパイルをホストマシン上で行うことで，マイクロコントローラよりも性能の高いマシンでより実行速度の速い機械語プログラムを生成可能になる．また，ホストマシン上でマイクロコントローラ上のプログラムのシンボル情報を記憶しておくことで，マイクロコントローラのメモリ上に配置するだけで実行が可能になる機械語プログラムを生成することができる．

本研究では提案手法を BlueScript の処理系として実装した．BlueScript は TypeScript 風の構文を有したマイクロコントローラ向けの言語である．BlueScript の開発環境は Notebook 風のインターフェースを持ち，ユーザはこの画面から対話的にプログラムを書くことができ

る．BlueScript の処理系は受け取ったソースコードを C 言語に変換し，既存のコンパイラを使用して機械語プログラムを生成後，ホストマシン上で動的リンクを行う．生成された機械語プログラムは Bluetooth 経由でマイクロコントローラに送信され実行される．

BlueScript の処理系を評価するために，本研究では 3 つの実験を行った．1 つ目の実験では BlueScript の対話性が維持されていることを調べるために，対話的にプログラムを追加していく micro:bit のチュートリアルプログラム [1] を使用して BlueScript と BlueScript でプログラムの追加にかかる時間を計測した．2 つ目の実験では Marr らの Cross language compiler benchmarking[2] と ProgLangComp[3] のマイクロベンチマークを使用して BlueScript と MicroPython，C 言語でプログラムの実行時間を測定した．3 つ目の実験では BlueScript と MicroPython のランタイムの大きさを調べた．実験で比較の際に使用した MicroPython は現在よく使われているマイクロコントローラ向けに最適化された Python の処理系である．

本研究の貢献は次の 3 つである．

- 動的コンパイラをホストマシンにオフロードするような新しいマイクロコントローラ向けの言語処理系のアーキテクチャを示したこと．
- 提案手法を BlueScript の言語処理系で実装したこと．
- 実装した言語処理系の実行速度とプログラムの変更にかかる時間，ランタイムのバイナリサイズを計測し提案手法の有効性を示したこと．

第 2 章

対話的な開発環境とプログラムの実行速度

2.1　マイクロコントローラ

マイクロコントローラとはその名の通り極小のコンピュータである．多くの場合，一つのボードに CPU，RAM，Flash，I/O 機器などが全て搭載されている．通常のコンピュータよりも安価で小型である代わりに，実行速度が遅く使用できるメモリ量も少ないなどの特徴がある．マイクロコントローラは主に産業用機器や医療用機器，家庭用機器などの制御が必要な製品に組み込まれて使われる．

また，安価で小型であることから，ディスプレイやマイク・ボタンなどと組み合わせて教育用に使用されることも多い．例えば，Raspberry Pi や micro:bit，ESP32 などが挙げられる．こういった教育用のマイクロコントローラは小型でありながら，ある程度の汎用性を保つために産業用のものよりも CPU やメモリが高性能であることが多い．

2.1.1　マイクロコントローラの構成

マイクロコントローラの CPU やメモリ構成は製造会社によって異なっている．

プロセッサはメモリ空間が比較的小さいことから 16 ビットプロセッサまたは 32 ビットプロセッサであることが多い．CPU のアーキテクチャは低消費電力であることで知られる ARM アーキテクチャが使われることが多いが，その他にも Xtensa，PowerPC，Intel アーキテクチャなどがある．

一度プログラムを書き込んだらその後プログラムを実行中に変更することがないため，ハーバードアーキテクチャを採用しているマイクロコントローラが多い．このアーキテクチャは命令を読み出す用のバスとデータを読み書きするためのバスが分かれていることが特徴である．CPU が命令とデータを同じ信号線を使って読み書きするノイマン型アーキテクチャに比べて命令の読み出しとメモリの読み書きが競合することがないため高速である．

メモリ構成については RAM やフラッシュメモリなど，いくつかの種類のメモリを組み合わ

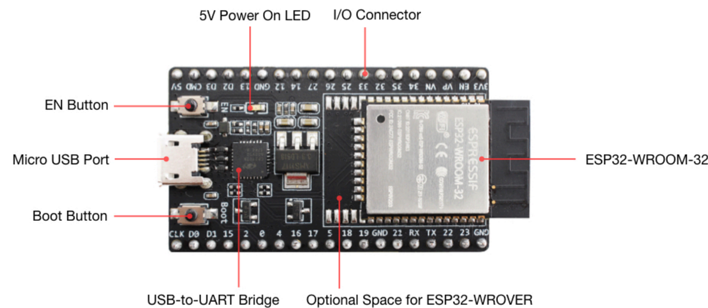

図 2.1. ESP32[*1]

せて使用している．RAM とは Random-access memory の略であり，バイト単位で読み書きが可能で比較的高速にメモリアクセスが可能なメモリである．電源が落ちるとデータが消えてしまうため短期記憶領域として使用される．フラッシュメモリ不揮発性メモリであり，RAM に比べて安価で電源が切れてもデータが消えてしまわない代わりに，データへのアクセス速度が遅く，ブロック単位でしか消去できない．主にプログラムを置くのに使用される．

2.1.2　ESP32

具体的なマイクロコントローラの例として ESP32 を紹介する．ESP32 は Espressif Systems が開発するマイクロコントローラのシリーズである．Xtensa single-dual-core 32bit LX6 マイクロプロセッサを持つ．CPU の周波数 240MHz である．メモリは内部メモリとして RAM を，外部メモリとしてフラッシュメモリを持つ．外部メモリとして PSRAM が付属しているデバイスもある．RAM の大きさは 520KiB であり，実行時に一時的にデータを置くために使われる．Flash の大きさは 4MiB または 8MiB であり，主にプログラムを配置するのに使用される．PSRAM は Flash よりはアクセススピードが速い外部接続された RAM であり，バイト単位でアクセスが可能であるが電源を切るとデータが消えてしまうため，カメラを使用した際の画像データを置くためなどに使用される．外部メモリは SPI 通信を使用してデータのやり取りをするため内部に配置された RAM よりも格段にアクセスが遅い．そのため，ESP32 では一部の RAM をキャッシュとして使用することでアクセススピードの高速化を図っている．

RAM は IRAM, DRAM, D/IRAM の三種類の領域に分類される．IRAM は Instruction-bus が繋がった領域であり，ここに置かれたデータは実行可能であるが，読み込みと書き込

[*1] Copyright 2021-2023 Espressif Systems (Shanghai) CO LTD

みはできない．DRAM は Data-bus が繋がった領域であり，ここに置かれたデータは実行することはできないが，読み込みと書き込みが可能である．D/IRAM は Instruction-bus と Data-bus の両方が繋がった領域であり，データを実行可能でかつ読み込みと書き込みも可能である．Flash に置かれたプログラムは IRAM の一部にマップされて実行される．

ESP32 は標準で無線通信機能として WiFi と Bluetooth を備えている．また，I/O インターフェースとして GPIO，PWM，SPI などを備えている．

2.2　マイクロコントローラの開発における対話的な開発環境の必要性

マイクロコントローラのプログラムの開発には対話的な環境がない C 言語などで行われることが多い．しかし，対話的な開発環境がない場合小さな変更に対しても毎回プログラムをビルドし直してフラッシュに書き込まなければならないため，マイクロコントローラ向けアプリケーションの開発は時間のかかるものとなる．本節では C 言語を用いた開発方法について述べた後，その問題点について具体的な例を使用して説明する．

2.2.1　C 言語による開発方法

マイクロコントローラ向けのアプリケーションを開発するよく使われる方法として C 言語で書く方法がある．多くの場合マイクロコントローラの製造メーカーが対象のマイクロコントローラ専用の OS やデバイスドライバーを提供しているため，それらを使用して開発を進める．

例えば，C 言語を用いて ESP32 上で動作するアプリケーションを開発する手順は次のようになる．

1. ホストマシン上で任意のエディタを使用して C 言語プログラムを書く．
2. ESP32 向けの GCC で対象の ESP32 上で直接実行可能な機械語にコンパイルする．
3. ホストマシンと ESP32 のデバイスをシリアルケーブルで繋ぎ，ESP32 のフラッシュにコンパイル済みのイメージを書き込む．
4. リセットボタンを押してプロブラムの実行を開始する．

一度書き込んだプログラムを変更する場合は，C 言語で書かれたプログラムを変更しもう一度 GCC によるコンパイルとシリアルケーブル経由でのフラッシュへの書き込みを行う．

2.2.2　C 言語による開発の問題点

C 言語による開発では，2.2.1 節で述べたようにプログラムの変更のたびにコンパイルとフラッシュへの書き込みをやり直さなければならない．この動作は，時間のかかるものであるため，小さな変更の度に行うと非常に煩雑である．

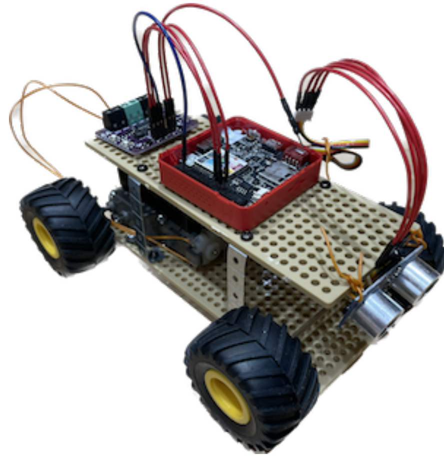

図 2.2. マイコンカー

例えば，図 2.2 にあるようなマイコンカーを作成することを考える．このマイコンカーはマイクロコントローラによって制御されるおもちゃの車である．先頭にはカメラを搭載しており，マイコンからの信号で写真を撮影しデータを返す．また，車輪を四つ備えており後方の二つの車輪はモータはそれぞれ別の AC モータにつながっている．この二つのモータの回転速度を制御すればマイコンカーは前後・左右に移動することが可能である．

プログラム 2.1 は C で書かれたマイコンカーのプログラムである．このプログラムを実行すると，マイコンカーは赤い物体を認識したら止まり認識しなければ前に進むという動きをする．16 行目で take_picture をして画像データを受け取った後，17 行目で detect_red_object 関数を呼び出し返り値が true であれば set_speed 関数に 0 を渡し，false であれば set_speed 関数に 50 を渡している．take_picture はカメラに信号を送って写真を撮影し，画像データ返す関数である．set_speed 関数は後輪に繋がったモータの回転数を調節するための関数である．1 行目から 12 行目に書かれている detect_red_object 関数は画像データを受け取って赤い物体があるかどうかを判定する関数である．detect_red_object 関数では，画像データの各セルについて Hue の値を計算し，計算した Hue の値と赤い色の HUE の値 (RED_HUE) の差が閾値以下であれば赤いセルであると判断して red_count 変数を 1 増やしている．全てのセルの Hue の値を確かめ終わった後に，もし赤いセルの数の合計値が閾値 (BOADER) 以上であれば true を返している．

前に進む際のモータの回転速度が早すぎると赤い物体に衝突してしまう可能性がある．このモータの回転速度は床からの摩擦などの影響を受けるため，実機でプログラムを動かして動きを見ながら調節する必要がある．モータの回転速度はプログラム 2.1 の 20 行目で set_speed に渡す値を変えることで調節可能である．

この調節を C 言語での開発環境で行う場合，20 行目の値を変更する度にプログラムをコン

```
1  bool detect_red_object(int32_t* image_data, int red_thres) {
2      int red_count = 0;
3      for (int row = 0; row < HEIGHT; row++) {
4          for (int col = 0; col < WIDTH; col++) {
5              int32_t cell_data = image_data[WIDTH * row + col];
6              int32_t hue = get_hue(cell_data);
7              if (hue > 0 && (abs(hue - RED_HUE) < red_thres || 253 - abs(
                   hue - RED_HUE) < red_thres))
8                  red_count++;
9          }
10     }
11     return red_count > BOADER;
12 }
13
14 int main() {
15     while (true) {
16         int32_t* image_data = take_picture();
17         if (detect_red_object(image_data, 10))
18             set_speed(0);
19         else
20             set_speed(50);
21
22         wait_ms(500);
23     }
24 }
```

Listing 2.1. 赤い物体を認識するプログラム

パイルし直して全てのイメージをフラッシュに書き込まなければならないため非常に煩雑である．例えば，このプログラムを実際にコンパイルして ESP32 にシリアルケーブル経由で書き込んだところ，約 7 秒の時間がかかった．一箇所調整するためだけに 7 秒もの時間が取られると，開発効率が大幅に落ちてしまうことは想像に難くない．

さらに，シリアルケーブル経由での書き込みも開発を煩雑にする．マイコンカーの例では，修正を加える度に毎回マイコンカーとホストマシンをシリアルケーブルで繋ぎ直さなければならない．そうでなければ，常にホストマシンとマイコンカーをシリアルケーブルで繋ぎっぱなしにして，ホストマシンを持ってマイコンカーを追いかけ回さなければならない．

2.3 対話的な開発環境と実行速度

前章で述べた問題点を回避するために，対話的な開発環境を提供している言語とその処理系を使用するという方法がある．対話的な開発環境では実行中に短い時間でプログラムの追加・

変更ができるため，小さな変更による調節が必要な開発も容易に行える．しかしながらマイクロコントローラ向けの対話的な開発環境は実行速度が遅いため，限定的にしか使用されていない．

2.3.1　対話的な開発環境

対話的な開発環境とは，その名の通りプログラムの実行結果を見ながら少しずつプログラムを追加していけるような開発環境である．代表的なものに Python 向けの対話的な開発環境である JuPyter Notebook や JavaScript のコンソールなどがある．本研究では対話的な開発環境を次の二つの条件を満たす開発環境であるとする．(1) 前の実行の結果を残したままプログラムの追加を行い，その結果を別の計算に使用することができる．(2) 十分に速い時間でプログラムを追加することができる．

2.3.2　MicroPython を使用した対話的な開発

マイクロコントローラ向けの言語処理系で対話的な開発環境を提供しているものはいくつかあるが，その中でも代表的なのが MicroPython である．MicroPython はマイクロコントローラ用に最適化された Python の言語処理系である．MicroPython は WebREPL という，ブラウザのページから対話的にプログラムを入力して実行する方法を提供している．WebREPL を使用して開発を行う手順は以下のようになる．

1. シリアルケーブルで開発に使用中のホストマシンと対象のマイクロコントローラを繋ぎ，MicroPython ランタイムを書き込む．
2. ターミナルからシリアルケーブル経由で対話的にプログラムが書き込める REPL 画面を開き，WiFi の設定をする．
3. WebREPL 画面を開き，マイクロコントローラと接続をする．
4. REPL 画面からプログラムを書き込んで実行することを繰り返す．

MicroPython の WebREPL 画面からの対話的な開発環境を使用すれば，小さな変更に対して毎回プログラムコンパイルしてフラッシュに書き込み直す必要がなく，効率的に開発を行える．さらに，無線で開発が行えるのでホストマシンとマイクロコントローラを常に繋いで置く必要もない．

プログラム 2.2 はプログラム 2.1 のマイコンカーのプログラムを MicroPython の WebREPL を使用して書いたものである．この例では set_speed に渡す値を speed 変数に入れておき，新しいスレッドを作成して loop 関数を実行している．loop 関数ではプログラム 2.1 の main 関数と同様に赤い物体を検知して進むかどうかを判断することを繰り返している．このような対話的な開発環境では，モータの回転速度を変更したければプログラム 2.2 にあるように新しい行で speed 変数の値を書き換えるだけで良いため，小さな変更の度にプログラムのビルドと書き込みを行う必要がなく効率的に開発を行える．

```
1  >>> def detect_red_object(image_data, red_thres):
2      ... red_count = 0
3      ... for row in range(HEIGHT):
4      ...     for col in range(WIDTH):
5      ...         cell_data = image_data[WIDTH * row + col]
6      ...         hue = get_hue(cell_data)
7      ...         if hue > 0 and (abs(hue - RED_HUE) < red_thres or 253 -
       abs(hue - RED_HUE) < red_thres):
8      ...             red_count += 1
9      ... return red_count > BOADER
10 >>> speed = 50
11 >>> def loop():
12     ... while(True):
13     ... image_data = take_picture()
14     ... if (detect_red_object(image_data, 10)):
15     ...     set_speed(0)
16     ... else:
17     ...     set_speed(speed)
18     ... wait_ms(500)
19 >>> _thread.start_new_thread(loop, ())
20 >>> speed = 100
```

Listing 2.2. 赤い物体を認識するプログラム

2.3.3 対話的な開発環境の実行速度

マイクロコントローラ向けの対話的な開発環境は実行速度が遅いという問題点がある．現在使用されているマイクロコントローラ向けの対話的な開発環境は，ホストマシンから開発者が書いた文字列を受け取り，それを解釈して実行するという実行方式をとっている．この方法では，マイクロコントローラのメモリが限られているため十分な最適化を行う言語処理系を搭載することができない．そのため，マイクロコントローラ向けの対話的な開発環境は，C 言語で開発する場合のようにホストマシン上で機械語に変換する方式と比べて実行速度が大幅に遅くなってしまう．

実際，カメラから渡される画像の大きさを 240x240 pixel に設定してプログラム 2.2 の detect_red_object 関数を一回実行するのにかかる時間を計測したところ，C 言語では 0.45 秒であったのに対し MicroPython では 11.5 秒もの時間がかかった．

表 2.1. V8 と Ruby の処理系のバイナリサイズ

言語	動的コンパイラあり (MB)	動的コンパイラなし (MB)
V8	33	22
CRuby	5.8	4.4

2.4 動的コンパイラとメモリ使用量

対話的な開発環境を提供する言語処理系の実行速度を高速化する技術として動的コンパイルが知られている．しかしながら，動的コンパイラは多くのメモリを消費するためマイクロコントローラ上の言語処理系では採用されない．本節では動的コンパイラについて説明した後，実際に現在使われている動的コンパイラがどれくらいメモリを消費するかを見ていく．

2.4.1 動的コンパイルとは

動的コンパイラとは，Just-in-Time コンパイラとも呼ばれ，プログラムの実行速度を高速化するためにプログラムの実行開始後にプログラムの一部または全てをより効率的な形式 (多くの場合は機械語) に変換するシステムのことである [4]．多くの動的コンパイラは，プログラムの最大パフォーマンスをできるだけ高くしつつ，最大パフォーマンスに到達するまでの時間を短くすることを目標としている．そのため，プログラムを全部コンパイルすると時間がかかってしまうこと，プログラムのほとんどの部分は実行されないという考察から，頻繁に実行される部分だけをコンパイルすることが多い．頻繁に実行される部分の特定方法は大きく分けて二つある．一つは頻繁に呼び出されるメソッドを対象にした method JIT と呼ばれる方法であり，もう一つは頻繁に実行されるループなどのコードブロック群を対象とした Trace based JIT [5][6] と呼ばれる方法である．前者は平均的に良いパフォーマンスを出せるというメリットがある一方，実装が比較的難しくネストした関数が多いと実行速度が上がりにくいというデメリットがある．後者は比較的実装が簡単であり，頻繁に実行される小さいループがあると有効であるというメリットがあるが，JIT の中で使用されるメモリ量が増えやすいというデメリットがる．

動的コンパイラは多くの言語で導入されている．HotpathVM[7] や trace-JIT[8]，IBMJ9[9] は Java の動的コンパイラである．TraceMonkey[10] や V8[11] は JavaScript の動的コンパイラである．PyPy[12] は Python の動的コンパイラである．YJIT[13] は Ruby の動的コンパイラである．

2.4.2 動的コンパイラのメモリ使用量

本研究では，JavaScript の言語処理系である V8 と Ruby の言語処理系である CRuby で使用されている動的コンパイラのメモリ使用量を調べた．

V8 は Google が開発するオープンソースの JavaScript 処理系である．Chrome ブラウザの

中や Node の中で使用されている．V8 に含まれている動的コンパイラの名前は Turbofan である．本研究では arm64 向けの V8 の大きさを以下のようにして計測した．

1. V8 のリポジトリ https://chromium.googlesource.com/v8/v8 から V8 のプログラムを `clone` する．
2. ビルドツールを使用してビルド．この際にビルドに含めるものを支持するオプションを指定する．
3. `du` コマンドを使用してビルド結果が含まれているディレクトリ内の `d8` イメージの大きさを調べる．`d8` は V8 の開発者用のシェルである．

動的コンパイラありの場合の V8 の大きさを測る場合には手順 2 でオプションとして `v8_enable_turbofan=true` と `v8_enable_webassembly=false` を指定した．動的コンパイラなしの場合の V8 の大きさを測る場合には手順 2 でオプションとして `v8_enable_turbofan=false` と `v8_enable_webassembly=false` を指定した．どちらの場合にも `v8_enable_webassembly=false` を指定した理由は Wasm コンパイラが Turbofan なしでは動かないためである．

CRuby はまつもとゆきひろによって開発が始められた Ruby の最も代表的な言語処理系である．CRuby はいくつかの動的コンパイラの実装を持つが，その中でも最新のものが YJIT である．YJIT[13] はネイティブコードにコンパイルするベーシックブロックを徐々に増やしていく Lazy Basic Block Versioning 方式で動的コンパイルを行っている．本研究では以下の方法で YJIT の大きさを計測した．

1. CRuby のリポジトリ https://github.com/ruby/ruby から CRuby のプログラムを `clone` する．
2. ビルドツールを使用してビルド．この際にビルドに含めるものを支持するオプションを指定する．
3. `du` コマンドを使用して `build` ディレクトリの `ruby` イメージの大きさを調べる．

YJIT ありの場合の CRuby の大きさを調べるためには，手順 2 でオプションとして—`-enable-yjit` を追加した．YJIT なしの場合の CRuby の大きさを調べるためには，手順 2 でオプションとして—`-disable-yjit` と—`-disable-rjit` を追加した．

表 2.1 は V8 と CRuby の動的コンパイラを含めた場合と含めなかった場合の大きさを記述したものである．この表から V8 では動的コンパイラを含めると大きさが 1.5 倍になることがわかる．また，CRuby では動的コンパイラを含めると大きさが約 1.3 倍になることがわかった．これらの結果から，動的コンパイラを搭載すると言語処理系の大きさが大幅に増えることがわかった．

2.5　関連研究

マイクロコントローラなどの組み込みシステム向けに，メモリ消費の小さい動的コンパイラを提案する研究はいくつか行われている [14][15][16][17][18]．例えば，Yuan Zhang et al. 2012[16] は Android 上で動作する Java の動的コンパイラを提案している．しかしながら，対話性に焦点を当てた研究はなされていない．また，組み込みシステム上の限られたメモリの中での動的コンパイルによるパフォーマンス向上は数倍程度にとどまっている．

その他にも，AOT Compiler (Ahead-Of-Time Compiler) を使用して，マイクロコントローラ向けの言語処理系の実行速度を向上させる研究もなされている．例えば，StaticTypeScript[19] はマイクロコントローラ向けの TypeScript の言語処理系の実行速度を高速化する研究である．StaticTypeScript は TypeScript 風の構文を有した言語であり，C++ に変換後一括コンパイルされる．アプリケーションの開発者はブラウザ上でソースコードを書き，ブラウザのボタンを押すことでコンパイル済みの実行ファイルを手にいれることができる．StaticTypeScript の開発環境はブラウザ上でマイクロコントローラの動きをシミュレートできるものの，実機での対話的な開発には対応していない．

マイクロコントローラの開発環境を改善する研究もなされている．Warduino はマイクロコントローラ上で動く WebAssembly の言語処理系である．アプリケーション開発者は Python などの高級言語で開発を行い，WebAssembly に変換して実行することができる．開発を容易にするために Warduino はデバッガを提供している．開発者は予めブレークポイントを指定しておき，そのブレークポイントでスタックや変数の中身などの実行に関わる状態を見ることができ，プログラムの更新もすることができる．Warduino は実行中にブレークポイントでプログラムの更新が可能であるが，マイクロコントローラ向けの言語処理系を高速化する研究ではなく，プログラムの実行速度は MicroPython に劣る．

第 3 章

提案：動的コンパイラをホストマシンにオフロードする．

3.1　動的コンパイラのオフロード

本研究では，マイクロコントローラのメモリを圧迫することなく，対話的な開発環境の実行速度を向上させる方法として，動的コンパイルのオフロードを提案する．MicroPython などの対話的な開発環境を提供する言語処理系は，MicroPython などの対話的な開発環境を提供する既存の言語処理系は，文字列を受け取ってから実行に至るまでの動作を全てマイクロコントローラ上で行っている．この方法ではマイクロコントローラが搭載しているメモリ量が限られているため，十分な最適化が行えない．特に対話的な処理系を高速化するための典型的な技術として知られている，動的コンパイルは前章で述べたように必要なメモリ量が多くマイクロコントローラ上に搭載するのは現実的ではない．

本論文では動的コンパイルをマイクロコントローラ上から開発に使用中のホストマシンにオフロードする方法を提案する．近年は個人の持つノートパソコンの性能も上がってきており，マイクロコントローラに比べて使用できるメモリ量も計算速度も格段に大きい場合が多い．そこで，ホストマシン上で動的コンパイルを行い十分な最適化をすることで，マイコンのメモリを圧迫することなくプログラムの実行速度を高速化する．

動的コンパイルは一般的に実行中にホットスポットを見つけてより実行速度の速い形式にコンパイルし直すことを指すが，ここではその前段階として，対話的に入力されたプログラム片を実行可能な機械語に変換する動的コンパイルをオフロードの対象とする．対話的に入力されたプログラム片を動的コンパイルする際のプログラム片の受理から実行までの流れは次のような手順で進む.

1. 対話的に入力されたプログラム片を受理.
2. 対象のデバイスで実行可能な機械語に変換.
3. シンボルのアドレスをリンク.
4. ロード.

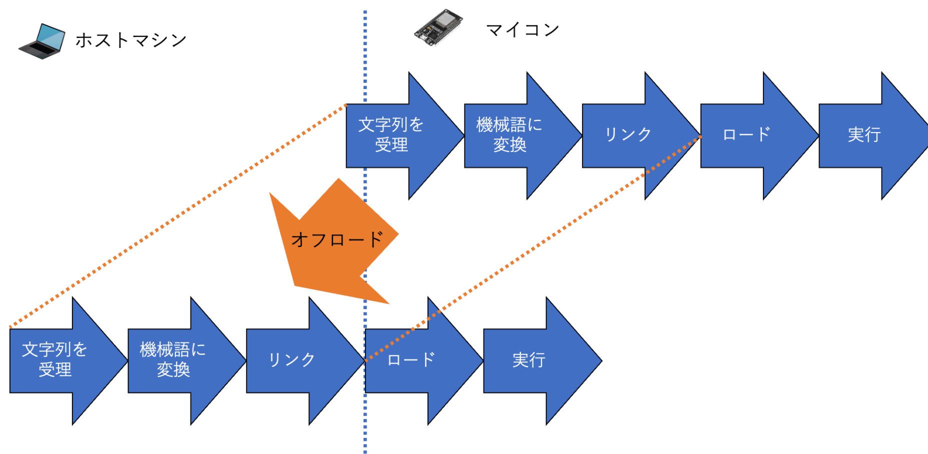

図 3.1. 動的コンパイラのオフロード

5. 実行．

シンボルとは機械語プログラム中でデータやプログラムのアドレスにつけたラベルのことであり，多くの場合変数名や関数名がそのまま使用される．

本論文で提案するのはこれらの手順 1-5 のうちの手順 1-3 のオフロードである．まず，ホストマシン上で手順 1-3 を行い実行可能な機械語プログラム片を生成する．その後，生成されたプログラム片をホストマシン上に送り，あらかじめ決められた場所に配置することで実行可能になる．

ホストマシン上でリンクまで行うことが可能なのは次の 3 つの前提があるためである．(1) ホストマシンはマイクロコントローラ上でシンボルが配置される位置を全て記憶している．(2) プログラムの実行中に，機械語プログラム中のシンボルの位置は変更されない．(3) 新しく対話的に追加されたプログラム片が実行中のプログラム中のシンボルへの参照を持つことはあるが，実行中のプログラムが新しく追加されたプログラム片の中のシンボルへの参照を持つことはない．以上三つの前提から，ホストマシンは記憶しているマイクロコントローラ上のアドレスを，新しく追加されたプログラム片を機械語に変換したものに埋め込むだけで実行可能な機械語プログラムが生成できる．

3.2　BlueScript とその開発環境

本研究では，提案手法の有効性を示すために，新しく BlueScript という言語を導入し，その言語処理系と開発環境を実装した．マイクロコントローラの対話的な開発を支援するために，BlueScript の開発環境はブラウザ上で使用可能な Notebook 風のインターフェースを持つ．BlueScript は TypeScript 風の構文を有した言語であり，BlueScript の処理系はガベージコレクションによって自動的にメモリを管理する．また，漸進的型付を採用しており，コン

図 3.2. BlueScript のインターフェース

パイル時と実行時の両方で型検査を行う．実行速度を向上させるために，BlueScript の言語処理系は対話的に追加されたプログラムを C 言語に変換して GCC を用いてコンパイルし動的リンクを行う．

BlueScript は Notebook 風のインターフェースを有しており，開発者は図 3.2 のようなブラウザの画面から実機を動かしながら対話的に開発を進めることができる．BlueScript を使用した開発では，開発者は以下のような手順で開発を進める．

1. シリアルケーブルを通してマイクロコントローラに BlueScript のランタイムのバイナリを書き込む．
2. ホストマシン上で BlueScript サーバーを立ち上げ, ブラウザから `http://localhost:3000/repl` にアクセスする．
3. 開いたブラウザのページ上 (図 3.2) で BlueScript プログラムを書き，セルの左側の実行ボタンを押す．このようにすることで，追加したプログラムをマイクロコントローラ上で実行することができる．
4. マイクロコントローラの実際の動きを見ながら，一番下のセルにプログラムを追加して実行することを繰り返す．

対話的な開発は Bluetooth 経由で行うため，最初に BlueScript 本体のバイナリを書き込んだ後はシリアルケーブルは必要ない．

3.3　動的コンパイラの設計と実装

図 3.3 は BlueScript の言語処理系の構成を表したものである．BlueScript は次の三つの要素から構成される．(1) ブラウザ上のエディタ．開発者はこの画面からプログラムを入力したり，ログを確認することができる．サーバから返されたバイナリを ESP32 のデバイスに送信する役割も担う．(2) ホストマシン上のサーバ．開発者が入力した BlueScript プログラムはこのサーバに送られ，実行可能なバイナリに変換されてブラウザに返される．(3)ESP32 のデバイス上のランタイム．受け取ったバイナリの実行やログの処理などを行う．

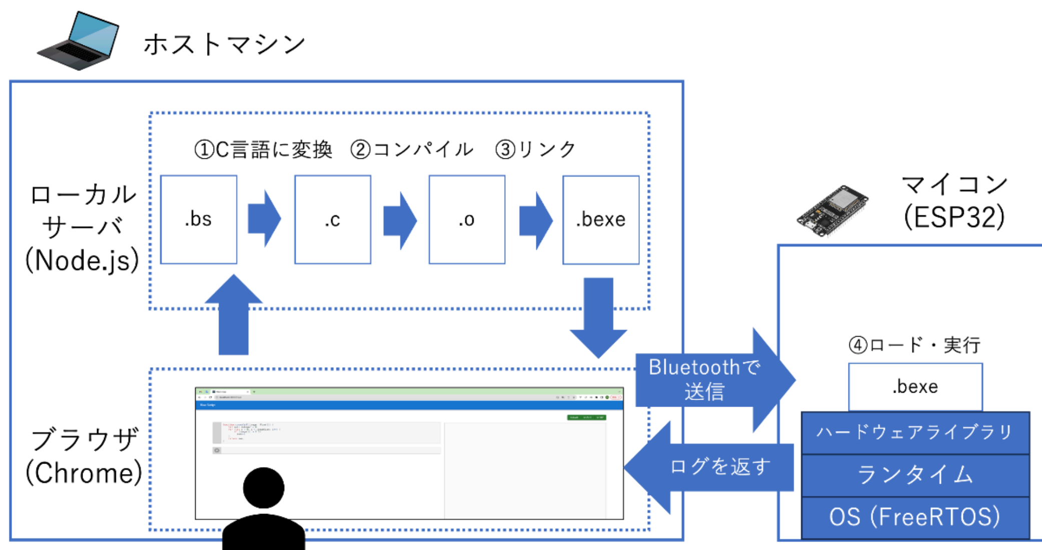

図 3.3. BlueScript の言語処理系の構成

開発者がプログラム片を入力して実行ボタンをから，実行が行われるまでの処理は以下のような手順で行われる．

1. ブラウザのページからローカルサーバに BlueScript のプログラム片が送られる．
2. サーバ上で BlueScript コード片が C 言語に変換される．(図 3.3 の 1)
3. GCC を使用して対象のマイクロコントローラデバイス (ESP32) で直接実行できるバイナリにコンパイルされる．この際リンクは行われない．(図 3.3 の 2)
4. BlueScript ランタイムやライブラリ関数，ユーザ定義関数がマイクロコントローラ上で配置されているアドレスなどがバイナリにリンクされる．(図 3.3 の 3)
5. 追加で定義された関数や変数のアドレスがサーバ上に保存される．
6. 生成されたバイナリがブラウザのページに送り返される．
7. バイナリがブラウザから Bluetooth 経由でマイクロコントローラに送信される．
8. バイナリがマイクロコントローラ上のランタイムによって実行される．(図 3.3 の 4)
9. ログが BlueScript ランタイムによってブラウザに返される．

本節では，対話的に入力されたプログラム片を BlueScript の言語処理系が ESP32 のマイクロコントローラ上で実行するまでの手順，つまり図 3.3 の 1 から 4 に当たる部分を順を追って説明していく．

C 言語に変換

対話的に入力された BlueScript プログラムはまずサーバ上で C 言語プログラムに変換される．図 3.3 の 1 の部分である．この際トップレベルの実行が必要な文は特別な名前のついた関数内に入れられる．

GCC でコンパイル

変換後に生成された C 言語プログラムは，ESP32 の製造メーカが提供している GCC コマンドを用いて ESP32 上で実行可能な Xtensa 命令セットの機械語プログラムにコンパイルされる．Xtensa は ESP32 に内蔵されているコアの名前であり，Xtensa 命令セットはその上で実行可能な命令セットである．図 3.3 の 2 に当たる部分である．この際-c オプションをつけてリンクはしない．また，最適化オプションは-O2 である．このようにすることで，未リンクの機械語プログラムの情報を含む ELF 形式のオブジェクトファイルが生成される．

リンク

次に，生成されたオブジェクトファイルから必要な情報を取り出し，対象のマイクロコントローラ上で実行可能になるようリンクする．図 3.3 の 3 に当たる部分である．取り出す情報は，次の四つである．(1) 機械語プログラム．(2)1 の機械語プログラムを実行するのに必要なデータ．(3) 機械語プログラムやデータの中でアドレスの埋め込みが必要な箇所を表した再配置テーブル．(4) 機械語プログラムやデータの中での関数や変数のオフセットが示されたシンボルテーブル．

さらに，すでにマイクロコントローラに書き込み済みの機械語プログラムがどのアドレスに配置されているかは，書き込みに仕様された ELF ファイルを解析することで知ることができる．これらの情報から，関数や変数を配置する場所を決め，(3) の情報をもとにアドレスを埋め込む．このようにして，マイクロコントローラの決められたメモリ上に配置するだけで実行可能になる機械語プログラムが生成できる．また，C 言語に変換した際に実行が必要なプログラムは特別な名前が付けられた関数内に入れられているため，その名前を使用してエントリポイントを見つける．

ロードと実行

このようにして生成された機械語プログラムがホストマシンによって Bluetooth 経由でマイクロコントローラに送信されると，マイクロコントローラ上で動作中の BlueScript ランタイムが受け取って配置と実行を行う．図 3.3 の 4 に当たる部分である．2.1.2 節で述べたように，ESP32 では実行可能でかつ実行時に読み書き可能な RAM の領域が限られているため，その領域に実行が必要な機械語プログラムを配置する．一方で実行が必要ないデータは実行できない領域に配置する．機械語プログラムと同時にホストマシンからエントリポイントも送られてくるため，エントリポイントにジャンプして実行を行う．

3.3.1 例を用いた説明

具体的な例を用いて説明していく．この例ではユーザがまず最初のセルでプログラム 3.1 のような関数を定義して実行ボタンを押した後，次のセルでプログラム 3.5 のような形で呼び出して実行することを想定している．最初のセルで定義される関数は与えられた引数同士を足す

```
1 function add(a:integer, b:integer) {
2   return a + b;
3 }
```

Listing 3.1. BlueScript のプログラム例：セル 1

```
1 static int32_t fbody_add(value_t self, int32_t _a, int32_t _b) {
2     { int32_t ret_value_ = (_a + _b); ; return ret_value_; }
3 }
4 struct func_body _add = { fbody_add, "(ii)i" };
```

Listing 3.2. C に変換後の BlueScript のプログラム例：セル 1

```
1  .text
2  <fbody_add>
3  00: 004136 // entry a1, 32
4  03: 234a  // add.n a2, a3, a4
5  05: f01d  // retw.n
6  07: 00 // ...
7
8  .data
9  <_add>
10 00: 00000000 // _add 構造体の一つ目の要素
11 04: 00000000 // _add 構造体の二つ目の要素
12
13 .rodata
14 00: 28696929 6900 // (ii)i. 文字列
```

Listing 3.3. GCC でコンパイル後の BlueScript のプログラム例：セル 1

```
1  .text
2  <fbody_add>
3  004011ec: 004136 // entry a1, 32
4  004011ef: 234a  // add.n a2, a3, a4
5  004011f1: f01d  // retw.n
6  004011f3: 00 // ...
7
8  .data
9  <_add>
10 00302214: ec114000 // _add 構造体の一つ目の要素: fbody_add が配置されるアド
       レス
11 00302218: 1c223000 // _add 構造体の二つ目の要素: (ii)i の文字列が配置され
       るアドレス
12
13 .rodata
14 0030221c: 28696929 6900 // (ii)i 文字列
```

Listing 3.4. リンク後の BlueScript のプログラム例：セル 1

だけの単純な関数である．

まずは最初のセルの実行ボタンが押された後に BlueScript の処理系内部で起きる動作を見ていく．プログラム 3.1 をブラウザ上の図 3.2 のような画面から入力して実行ボタンを押すと，C 言語のプログラムに変換されてプログラム 3.2 が生成される．BlueScript の処理系は 3.4.2 節で述べたように，関数の上書きを可能にするために関数を間接参照にしている．そのため，プログラム 3.2 では，実際の関数を定義した後，その関数のアドレスを構造体に格納している．

生成された C 言語プログラム 3.2 は次に ESP32 向けの GCC で Xtensa 命令セットを使用した機械語プログラムにコンパイルされる．コンパイルの際に-c オプションを用いることで，まだリンクはしていない状態のオブジェクトファイルを生成する．生成されたオブジェクトファイルの必要な部分を取り出したのが図 3.3 である．プログラム 3.3 内の.text，.data，.rodata はそれぞれのデータが配置される場所を示している．プログラム 3.3 の 3-6 行目に書かれているのが，プログラム 3.2 の fbody_add を Xtensa 命令セットの機械語にコンパイルしたものである．プログラム 3.3 の 10-11 行目に書かれている部分がプログラム 3.2 の_add 構造体のデータが格納される部分を表している．プログラム 3.3 の 14 行目に書かれている部分がプログラム 3.2 の"(ii)i"文字列を表している．

GCC でコンパイルされた機械語プログラムをマイクロコントローラ上のアドレスを用いてリンクされる．リンク時に埋め込むアドレスは，ESP32 上ででリンク済みの機械語が配置される予定の場所から算出する．2.1.2 節で述べたように，ESP32 では実行可能でかつ実行時に読み書き可能な RAM の領域が限られているため，その領域に実行予定の機械語である.text 領域のデータを配置する．一方で.data や.rodata のデータは実行する必要がないため，実行できない領域に配置する．ここでは実行可能でかつ実行時に読み書き可能な領域の最初のアドレスを 0x004011ec，実行できない領域の最初のアドレスを 0x00302214 とする．実行される可能性のあるデータである fbody_add 関数のプログラムはアドレス 0x004011ec から連続したメモリ領域に配置される．実行されないデータである_add 構造体のデータと文字はアドレス 0x00302214 から連続したメモリ領域に配置される．

これらのアドレスが決まると，コンパイル後の機械語にアドレスを埋め込めるようになる．アドレスを埋め込む必要がある場所は GCC で生成されたオブジェクトファイルの再配置テーブルを見ることでわかる．(再配置テーブルの情報は省略している．) 今回はプログラム 3.3 の 10 行目と 11 行目がである．ここに必要なアドレス，今回の場合は fbody_add 関数のアドレス 0x004011ec と"(ii)i"文字列のアドレス 0x0030221c を埋め込む．ESP32 上の機械語はリトルエンディアンで配置されるので，アドレスをリンク後の機械語はプログラム 3.4 のようになる．プログラム 3.4 では読みやすいように関数名やセクション名，アドレスやコメントを追加しているが実際に Bluetooth で送られて配置されるのは機械語の部分だけである．

ホストマシン上で生成された機械語 (プログラム 3.4) は Bluetooth 経由で ESP32 のマイクロコントローラに送信される．これらの機械語を受け取ったマイクロコントローラは適切な領域，実行するかもしれない機械語は 0x004011ec から始まる実行可能領域に，実行されないデータは 0x00302214 から始まる実行できない領域に配置する．

```
add(1, 2);
```

Listing 3.5. BlueScript のプログラム例：セル 2

```
extern struct func_body _add;

void bluescript_main2() {
    ((void (*)(value_t, int32_t, int32_t))_add.fptr)(0, 1, 2);
}
```

Listing 3.6. C に変換後の BlueScript のプログラム例：セル 2

```
.literal
0: 00000000

.text
<bluescript_main2>
0: 004136  // entry a1, 32
3: 000081  // l32r a8
6: 2c0c  // movi.n a12, 2
8: 0888  // l32i.n a8, a8, 0
a: 1b0c  // movi.n a11, 1
c: 0a0c  // movi.n a10, 0
e: 0008e0  // callx8 a8
11: f01d  // retw.n
```

Listing 3.7. GCC でコンパイル後の BlueScript のプログラム例：セル 2

```
.literal
004011f4: 14223000

.text
<bluescript_main2>
004011f0: 004136  // entry a1, 32
004011f3: 410200  // l32r a8
004011f6: 2c0c  // movi.n a12, 2
004011f8: 0888  // l32i.n a8, a8, 0
004011fa: 1b0c  // movi.n a11, 1
004011fc: 0a0c  // movi.n a10, 0
004011fe: 0008e0  // callx8 a8
00401201: f01d  // retw.n
```

Listing 3.8. リンク後の BlueScript のプログラム例：セル 2

以上が一つ目のセルにプログラム 3.1 を書き込んで実行ボタンを押した時に BlueScript の言語処理系の中で起こる動作である．次に，新しくセルを追加して，プログラム 3.5 のように定義した add 関数を呼び出すプログラムを書き，実行ボタンを押した時に BlueScript の処理系内部で起きる動作を説明する．

プログラム 3.5 を C 言語に変換すると，プログラム 3.6 が生成される．実行が必要なプログラム片は特別な名前がついた関数 (ここでは bluescript_main2) に入る．_add 関数は間接参照になっているので，プログラム 3.6 の 4 行目のように呼び出される．1 行目の extern キーワードから始まる文はコンパイルエラーを防ぐために挿入してある．

C 言語で書かれたプログラム 3.6 を-c オプション付きの GCC コマンドで ESP32 向けの機械語に変換するとプログラム 3.7 が生成される．プログラム 3.7 の 6 から 13 行目に書かれているのが関数 bluescript_main2 をコンパイルしたものである．プログラム 3.7 の 2 行目にある.literal 領域のデータは関数 bluescript_main2 内でデータをロードする際にロードするアドレスを置く場所として指定される．

このプログラムはセル 1 の機械語が配置された場所の続きの領域に配置される．実行される可能性のあるプログラムは 0x004011f4 = 0x004011ec + 8 に配置される．0x004011f3 は上で述べた，実行可能領域の最初のアドレスである．8 は fbody_add のプログラムによってすでに使用されたメモリ量である．実行されないデータは 0x00302222 = 0x00302214 + 14 に配置される．0x00302214 は上で述べた，実行できない領域の最初のアドレスである．14 はセル 1 の文字列や構造体によってすでに使用された領域である．

Xtensa 命令セットの制約上，.literal 領域のデータはそのデータを参照する命令の 262144 byte から 4 byte 前に配置しなければならないため，.literal 領域のデータは実行可能領域に配置し，続いて.text 領域のデータつまり bluescript_main2 の機械語を配置する．さらに，Xtensa 命令セットの仕様に則って，プログラム 3.7 の 2 行目と 7 行目にアドレスを埋め込む．こうして出来上がる機械語がプログラム 3.8 である．

また，今回は実行が必要なプログラムを含むため，実行のエントリポイントのアドレスもマイクロコントローラに教えなければならない．実行が必要なプログラムはすべて bluescript_main2 関数の内部に含まれているため，bluescript_main2 のアドレスである 004011f0 を教えれば十分である．

生成された機械語プログラムと実行のエントリポイントを Bluetooth 経由でホストマシンからマイクロコントローラに送信し，実行する．機械語プログラムの配置は，セル 1 の機械語プログラムを配置した場所の続きに置くだけである．プログラムの実行は機械語プログラムの配置後にプログラム void (*entrypoint)(void) = 0x004011f4; entrypoint(); のような形で呼び出せば実行することができる．

```
1 let a: any = 3;
2 a = 3.3;
3 let f: float = a; // Ok
4 let i: integer = a; // runtime type error
```

Listing 3.9. any 型を使用する例

3.4 BlueScript の言語仕様と実装

3.4.1 BlueScript の言語仕様

BlueScript の構文ははエディタなどの既存の開発環境をなるべく使用できるよう，TypeScript の主要な構文をそのまま流用している．一方でプログラムの実行速度を向上しランタイムのバイナリサイズを小さくするため一部 TypeScript とは異なった意味付けをしている．また，過度に動的なプロトタイプや `eval` などの機能はサポートしていない．モジュール機能やメソッドなど未実装の部分もある．変数宣言時のキーワードは `let` と `const` のみをサポートしている．

本節では BlueScript の特徴の中でも TypeScript と大きく異なる部分であるデータ型の意味付けの部分を説明していく．

プリミティブ型

BlueScript のプリミティブ型は integer，float，boolean，string, null であり，integer と float を静的に区別している．全て 32 bit で表現される．undefined はサポートしていない．

オブジェクト型

BlueScript ではプリミティブ型と後述する any 型以外の全ての型はオブジェクト型である．ユーザは TypeScript と同じようにクラスの宣言時にプロパティとコンストラクタを定義することができる．メソッドは未実装である．プロトタイプは持たない．

BlueScript は組み込みクラスとして Array 型を持つ．Array の宣言，初期化，要素へのアクセスは TypeScript と同様の方法で可能である．作成された Array オブジェクトへの操作は要素の代入以外は未実装である．

any 型

実行速度の向上とユーザの使いやすさを両立させるために，BlueScript は漸進的型付を取り入れている．漸進的型付とは 2006 年に Siek and Taha 2006[20] で初めて紹介された型システムで，静的型システムと動的型システムの良いところを組み合わせた型システムである．静的型システムは早期のエラー発見や効率的なプログラム実行，プログラムの読みやすさなどを提供する．動的型システムは実装の容易さと柔軟さを提供する．この両者を組み合わせた漸

```
1 let any_i: any = 24;
2 any_i += 1;
3
4 let any_str: any = "helloworld";
5 let str: string = "helloworld";
```

Listing 3.10. any 型を使用する例

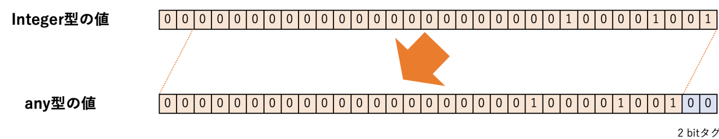

図 3.4. integer 型の値を any 型の値に変換する例

進的型システムでは，開発者が同じプログラム内で型アノテーションをつけたりつけなかったりすることで，どの程度静的型検査をやるかを決めることができる．漸進的型システムでは実行時まで型がわからない unknown 型を定義する．漸進的型システムにおける unknown 型を BlueScript で表すのが any 型である．BlueScript の any 型は次のような特徴を持つ．(1)any 型にノーテーションされた変数やプロパティには全ての型の値を代入することができる．(2)any 型の変数やプロパティに格納された値をその他の値に Cast する場合は，その変数やプロパティに格納されている値が Cast 対象の型にラベル付されていない場合ランタイムエラーになる．

例えば，プログラム 3.10 のようなプログラムが記述可能である．このプログラムでは，まず any 型の変数 a を宣言し，integer 型の変数を代入している．a は any 型の変数なので 2 行目のように float 型の値も代入可能である．2 行目で float 型の値を格納したので 3 行目で float 型の変数に any 型の変数に格納された値を代入することが可能である．一方で 4 行目のように integer 型の変数に代入しようとするとランタイムエラーになる．

関数

関数は引数と返り値の型が同じ場合のみ上書き可能である．アロー関数は宣言可能であるが，スコープの外側の変数の値はキャプチャしない．

3.4.2 BlueScript の言語処理系の実装

any 型の実装

BlueScript の any 型では実際のデータ型を見分けるために下位 2bit をタグとして使用している．00 が integer, 01 が float, 10 がオブジェクトへのポインタを表す．11 は使用されていない．boolean 型は integer としてタグ付けされる．null は null ポインタ，つまり値を表す部分が 0 でタグが 11 の値になる．

```
class Position {
  x: integer;
  y: integer;
  z: any;

  constructor(x: integer, y: integer) {
    this.x = x;
    this.y = y;
    this.z = null;
  }
}

let pos = new Position(12, 11);
pos.x += 1;
pos.z = 3;
```

Listing 3.11. クラスの定義と使用例

実行速度を向上させ，使用メモリ量を減らすために BlueScript では Boxing ではなく値の精度を 32 bit から 30 bit に落として，タグのために 2 bit 使用する方法を選択した．整数型の値からは上位 2 bit を奪っている．浮動小数点型の値からはできるだけ精度を保つために指数部から 2 bit 奪っている．any 型の値に対して演算を行う場合はタグをチェックして値が integer または float のタグ，つまり 00 または 01 を持っていれば一度元の型の値に戻し，演算を行なった後にもう一度タグづけする．そのため any 型の値の演算は integer 型や float 型の演算に比べてはオーバヘッドが伴う．

`string` や `array`，ユーザ定義クラスのインスタンスなど，ポインタを経由してアクセスされるデータは全て BlueScript のランタイム内では常に any 型の値として扱われる．後述するように，オブジェクトの配置は 4byte align なので下位 2bit をタグとして使用しても問題ない．

例として，プログラム 3.10 のようなプログラムが実行される場合を考える．プログラム 3.10 では 1 行目で値 24 が any 型の変数 `i` に代入される．この時，値 24 は integer 型であるため，図 3.4 のように下位 2bit をタグ 00 に使用できるよう値が変換される．3 行目のように演算が行われる場合はタグをチェックして元の値が integer であることを確認し，一度タグを取って演算を行なった後にもう一度タグづけする．上述のようにポインタを経由してアクセスされる値は全て any 型にした場合と同様の値が入るため，5 行目と 6 行目を実行した場合はポインタに対して同様のタグ付け演算が行われる．

クラスとオブジェクトの実装

BlueScript ではクラスを定義した場合，ガベージコレクション対象外の領域にクラスオブジェクトが作成される．メソッドは未実装である．クラスオブジェクトは作成されるインスタ

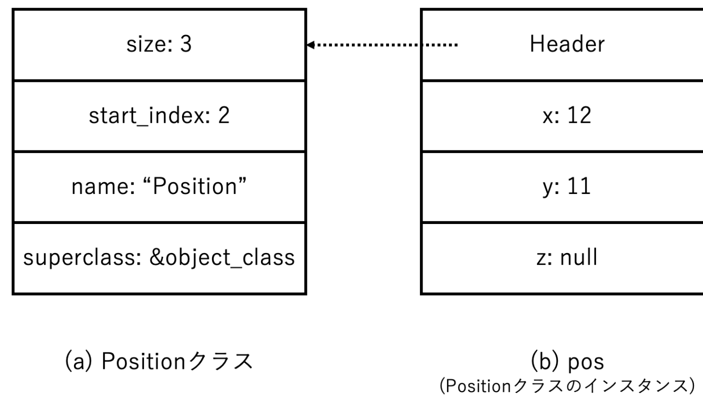

図 3.5. Class とオブジェクトの模式図

ンスオブジェクトの大きさ，最初の any 型の値を持つプロパティのインデックス，クラス名，スーパークラスへのポインタなどの情報をもつ．最初の any 型の値を持つ最初のプロパティのインデックスはプロパティが any 型の値を持つかどうかを見分けるために使用される．any 型の値でないプロパティがデータの中で前半に来るように，処理系は最初にプロパティの順番を並べ替える．

BlueScript ではインスタンスオブジェクトはヘッダとボディからなり，ガベージコレクション対象のヒープ領域に 4 byte align で配置される．ヘッダは 1 ワード (32bit) データで，上位 30 bit がクラスオブジェクトが配置されているアドレスを指し，下位 2 bit がガベージのマークビットとして使用される．プロパティの数や位置はクラス作成後に変更されないため，オブジェクトのプロパティにアクセスする場合はプロパティの場所をインデックスで使用して高速にアクセスが可能である．

例として，プログラム 3.11 のようなプログラムが実行される場合を考える．プログラム 3.11 の 1 行目から 11 行目が実行されると，図 3.5 の (a) ようなデータとコンストラクタ関数が作成される．`Position` クラスはプロパティを三つしか持っていないため，`size` は 3 である．最初の any 型の値でないプロパティである z のインデックスは 2 であるため，`start_index` には 2 が格納されている．BlueScript において全てのユーザ定義クラスはオブジェクトクラスのサブクラスであるため，`Position` クラスはスーパクラスへのポインタとして `object_class` へのポインタを持つ．

プログラム 3.11 の 13 行目を実行すると図 3.5 の (b) のようなようなインスタンスオブジェクトがガベージコレクション対象のヒープ上に作成される．ヘッダは `Position` クラスオブジェクトへのポインタを持つ．ヘッダの下位 2 bit はタグとして使用される．ボディには `x`，`y`，`z` のデータが格納される．14 行目のようにプロパティ x にアクセスする場合は，`x` がインスタンスオブジェクト内で格納されている場所が静的にわかるため，インデックス `0` を用いてアクセスが可能である．また，`x` のインデックス `0` が `Position` クラスの `start_index` のよりも小さいため，any 型の値でないことがわかり，タグをつけたり外したりせずに演算が可能

```
1 let str: string = "helloworld";
```

Listing 3.12. 文字列の定義例

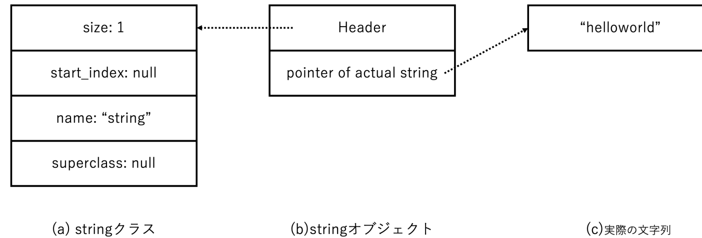

図 3.6. 文字列クラスとそのインスタンスの模式図

である．15 行目のように any 型のプロパティに値を代入する場合は，値 3 に対するタグ付けが必要である．

文字列の実装

BlueScript は string 型のデータを作成するためにビルトインクラスオブジェクトとして文字列クラスオブジェクトを持つ．文字列クラスオブジェクトは他のクラスオブジェクトと同様に，作成されるインスタンスオブジェクトの大きさ，最初の any 型の値を持つプロパティのインデックス，クラス名，スーパークラスへのポインタなどの情報をもつ．作成されるインスタンスオブジェクトの大きさは 1 である．最初の any 型の値を持つプロパティのインデックスは使用されないため `null` が格納されている．string 型はプリミティブ型であるためスーパクラスへのポインタが格納される場所には `null` が格納されている．

文字列クラスのインスタンスオブジェクトは他のインスタンスオブジェクトと同様にボディとヘッダからなる．ヘッダには文字列クラスオブジェクトへのポインタが格納されている．ボディは一つであり，実際の文字列データへのポインタが格納されている．文字列データはガベージコレクション対象外のヒープ領域に格納される．

例としてプログラム 3.12 のようなプログラムが実行される場合を考える．文字列クラスオブジェクトは図 3.6 の (a) のようなデータ構造をしている．プログラム 3.12 の 1 行目を実行すると，図 3.6 の (b) のようなインスタンスオブジェクトが作成される．このインスタンスオブジェクトは文字列クラスオブジェクトへのポインタをヘッダに持ち，`"helloworld"`文字列データへのポインタをボディに持つ．

配列の実装

BlueScript は要素の型が integer・float・boolean 以外の配列のデータを作成するために Array クラスオブジェクトを持つ．Array クラスオブジェクトの要素は BlueScript のランタイム内で全て any 型として扱われる．Array クラスオブジェクトは他のクラスオブジェクトと同様に，作成されるインスタンスオブジェクトの大きさ，最初の any 型の値を持つプロパ

```
1 let arr: any[] = [1, 0.2, "helloworld"];
2 arr[0] = "foo";
```

Listing 3.13. クラスの定義と使用例

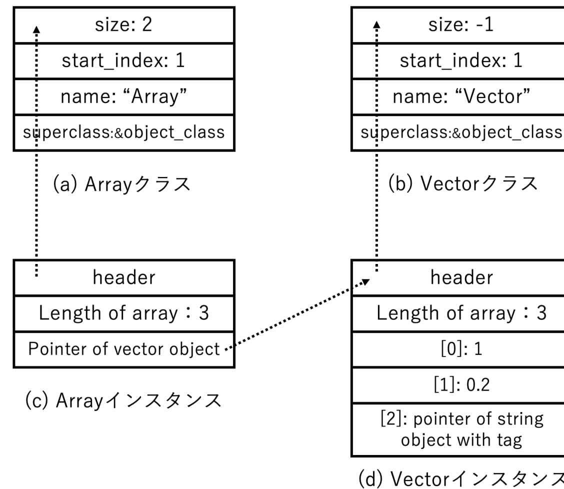

図 3.7. 配列を初期化した際に作成されるデータの模式図

ティのインデックス，クラス名，スーパークラスへのポインタなどの情報をもつ．Array クラスのインスタンスオブジェクトの大きさは 2 である．Array クラスの最初の any 型の値を持つプロパティのインデックスは 1 である．また，スーパクラスへのポインタとしてオブジェクトクラスオブジェクトへのポインタを持つ．

BlueScript は要素の数が固定の配列クラスを表すためにビルトインの Vector クラスを持つ．Vector クラスオブジェクトはユーザが直接操作することはできない．Vector クラスオブジェクトは他のクラスオブジェクトと同様に，作成されるインスタンスオブジェクトの大きさ，最初の any 型の値を持つプロパティのインデックス，クラス名，スーパークラスへのポインタなどの情報をもつ．作成されるインスタンスオブジェクトの大きさは要素の数 +1 であり，最初の any 型の値を持つプロパティのインデックスは 1 である．Vector クラスのインスタンスオブジェクトはボディの要素の一つ目に配列の要素の数を持ち，二つ目以降に実際の要素の値を持つ．また，スーパクラスへのポインタとしてオブジェクトクラスオブジェクトへのポインタを持つ．

Array 型の値を初期化すると，Array クラスのインスタンスオブジェクトと Vector クラスのインスタンスオブジェクトが作成される．Array 型の値が変数などに格納される場合は，Array クラスのインスタンスオブジェクトへのポインタを 2 bit タグ付けしたものが格納される．Array クラスのインスタンスオブジェクトボディの第一要素に配列の大きさを持ち，第二要素に Vector クラスのインスタンスオブジェクトへのポインタを持つ．

例としてプログラム 3.13 が実行される場合を考える．Array クラスオブジェクトと Vector クラスオブジェクトは図 3.7(a)・(b) のようなデータ構造をしている．プログラム 3.13 の 1

```
1 let arr: integer[] = [1, 5, 7];
2 arr[0] = 3;
```

Listing 3.14. 配列の定義と使用例

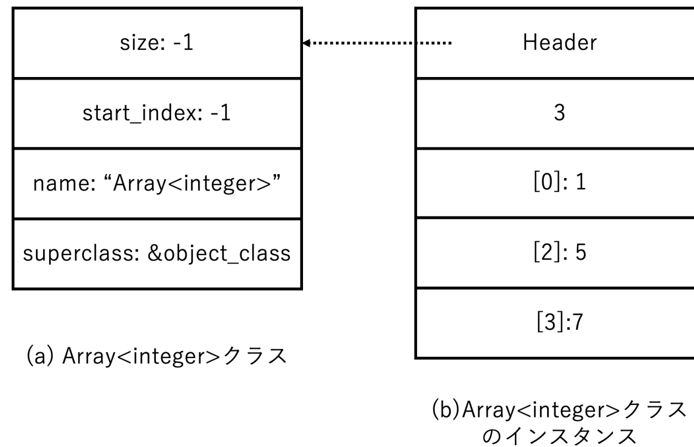

図 3.8. 配列を初期化した際に作成されるデータの模式図

行目が実行されると図 3.7 の (c)・(d) のようなオブジェクトが作成される．配列の長さは 3 であるため，(c) の Array インスタンスオブジェクトのボディの一番目の要素には 3 が入り，二番目の要素には Vector インスタンスオブジェクトへのポインタが 2 bit タグ付けされたものが入る．(d) の Vector インスタンスオブジェクトのボディの最初の要素には配列の長さである 3 が入り，それ以降に配列の要素の値が続く．プログラム 3.13 の 2 行目を実行すると，Vector インスタンスのボディの 2 番目の要素が文字列インスタンスオブジェクトへのポインタに書き換わる．

BlueScript は余計なタグ検査やタグの付け外しをなくしプログラムの実行を高速化するために，integer，float から成る配列の型 (`Array<integer>`, `Array<float>`) のデータを表す特別なクラスオブジェクトを用意している．`Array<integer>`型を例に取って説明する．`Array<integer>`クラスオブジェクトは他のクラスオブジェクトと同様に，作成されるインスタンスオブジェクトの大きさ，最初の any 型の値を持つプロパティのインデックス，クラス名，スーパークラスへのポインタなどの情報をもつ．`Array<integer>`クラスのインスタンスオブジェクトの大きさは配列の要素の数 `+1` である．`Array<integer>`クラスのインスタンスオブジェクトは内部にポインタを持たないため，最初の any 型の値を持つプロパティのインデックスは`-1` である．また，スーパクラスへのポインタとしてオブジェクトクラスオブジェクトへのポインタを持つ．

`Array<integer>`型の値を初期化すると，`Array<integer>`クラスのインスタンスオブジェクトのみが作成される．Array 型の値を初期化した場合と違い Vector クラスのインスタンスオブジェクトは作成されない．`Array<integer>`クラスのインスタンスオブジェクトは

```
1 const foo = () => 2;
```

Listing 3.15. const 付き関数宣言

```
1 function foo() {
2   return 2;
3 }
4 foo();
5
6 let goo: () => integer = foo;
```

Listing 3.16. 関数の定義と使用例

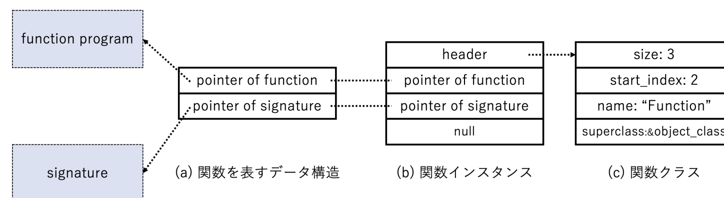

図 3.9. 関数周りのデータ構造

ヘッダに Array<integer>クラスオブジェクトへのポインタをもつ．ボディの最初の要素の値は配列の長さであり，二つ目以降に配列の実際の値が続く．配列の要素にアクセスする際にポインタを一つしか辿らなくて良いことと要素のタグのチェックが必要ないことから，Array<integer>への要素のアクセスは Array の要素のアクセスよりも高速である．

例としてプログラム 3.14 を実行する場合を考える．1 行目を実行すると図 3.8 の (b) のようなインスタンスオブジェクトが作成される．ボディの一つ目の要素には配列の長さである 3 が格納されており，それ以降に実際の値が続く．

関数の実装

BlueScript の処理系では，関数は後から上書きできるように間接参照になっている．関数が定義されると，ガベージコレクション対象外の領域に実際の関数プログラムへのポインタと関数のシグネチャを持ったデータが作成される．シグネチャは関数の引数と返り値の型を表している．関数が変数に代入されたり引数に渡されたりすると，ここで初めて関数オブジェクトが作成される．この関数オブジェクトは関数が変数に代入されたり引数に渡されたりする度に作成される．関数オブジェクトはインスタンスオブジェクトと同様にヘッダとボディから成り，ヘッダは関数クラスオブジェクトを指す．ボディの最初の要素は実際の関数プログラムへのポインタであり，二番目の要素はシグネチャへのポインタを持つ．三番目の要素は関数があるクラスのメソッドである場合に this オブジェクトを指すのに使用される予定であるが，メソッドは未実装であるため今は空欄である．例外として，図 3.15 のように const 付きで宣言され

た場合のみ，関数は間接参照にならない．

関数クラスオブジェクトは BlueScript 処理系のビルトインクラスオブジェクトであり，他のクラスオブジェクトと同様に，作成されるインスタンスオブジェクトの大きさ，最初の any 型の値を持つプロパティのインデックス，クラス名，スーパークラスへのポインタなどの情報をもつ．オブジェクトの大きさは 3 で最初の any 型の値を持つプロパティのインデックスは 2，スーパクラスのポインタはオブジェクトクラスオブジェクトへのポインタである．

例としてプログラム 3.16 を実行する場合を考える．1 行目から 3 行目を実行すると図 3.9 の (a) が作成される．4 行目を実行すると，図 3.9 の (a) に含まれる関数ポインタを辿って関数が実行される．6 行目を実行した時初めて，図 3.9 の (b) が作成される．図 3.9 の (b) の関数インスタンスオブジェクトのボディの一つ目と二つ目の要素は図 3.9 の (a) のデータ構造の一つ目の値と二つ目の値と同じである．関数インスタンスオブジェクトのヘッダは関数クラスオブジェクトへのポインタを含む．

ガベージコレクション

BlueScript はマーク&スイープ方式のガベージコレクタを採用している．生きているオブジェクトはルートセットからスタックを利用して深さ優先探索でたどる．生きているオブジェクトとはルートセットから到達可能なオブジェクトのことであり，逆に死んだオブジェクトとはルートセットから到達不可能なオブジェクトのことである．マークビットにはインスタンスオブジェクトのヘッダの下位 1 ビットを使用している．スタックオーバフローした際の対応は The Garbage Collection Handbook[21] で提案されている方法と同様の方法を用いている．

ルートセットはリンクトリストで管理している．関数が呼ばれると関数の中で新たにシャドウスタックを作成し全体のルートセットの末尾に繋げ，関数の呼び出しが終わる直前で関数内で作成したシャドウスタックを削除している．ルートセットには，対象のスコープ内で作成された全ての any 型の値，つまりポインタを含むかもしれない全ての値を登録する．

BlueScript はマイクロコントローラ上で動作することを想定して作成されているため，ガベージコレクション中に発生した割り込みにも対応している．割り込みとは，周辺機器などからの要求で CPU が現在動作中のプログラムの実行を一時的に中断して，要求があった先の別のプログラムの実行を行うことである．多くの場合，割り込みの動作は極短時間で終わり，また元のプログラムに戻ってくる．この割り込みが，ガベージコレクションの実行中に発生した場合，生きているオブジェクトが回収されてしまう可能性が出てくる．

話を分かりやすくするために，ここでオブジェクトの三色抽象化 [22] を導入する．オブジェクトの三色抽象化とはガベージコレクション中のオブジェクトの状態によってオブジェクトに白色・灰色・黒色の色を付けることである．白色オブジェクトはまだルートセットから辿られていないオブジェクトである．灰色オブジェクトはルートセットから到達されたが，その子オブジェクトはまだ辿っていない状態のオブジェクト，つまりスタックに積まれた状態のオブジェクトである．黒色オブジェクトはルートセットから到達されて，かつその子オブジェクトも辿り終わったオブジェクトである．オブジェクトの色を見分けるために，BlueScript ではマークビットとインスタンスオブジェクトの下から 2 番目のビットを合わせて使用している．

ガベージコレクション中に割り込みが発生した場合に，対応が必要なのは黒色オブジェクトに対して，白色オブジェクトのポインタが書き込まれた場合である．もし何も対応しなければ，この白いオブジェクトはルートセットから辿られず，スイープフェーズで生きているのに回収されてしまう可能性がある．そこで BlueScript では，オブジェクトに対して any 型の値の書き込みを行う動作に対して，次の二つの条件が揃った場合にポインタの指す白いオブジェクトを灰色にする (つまりスタックに積む) ようなライトバリアを挿入している．(1) ガベージコレクション中に発生した割り込みの中の動作である．(2) any 型の値が白色オブジェクトへのポインタである．

第 4 章

実験

本章では BlueScript の性能を評価するために行った三つの実験の結果を示す．一つ目の実験では，対話性を調べるためにプログラムを変更するのにかかる時間を BlueScript, MicroPython，C 言語で比較した．二つ目の実験では，幾つかのマイクロベンチマークを用いて BlueScript，MicroPython，C 言語の実行速度を比較した．三つ目の実験では，ランタイムのバイナリサイズを BlueScript と MicroPython で比較した．

4.1　実験環境

実験にはマイクロコントローラモジュールとして M5Stack Fire を，ホストマシンとして Macbook Pro を使用した．Macbook Pro は Apple M1 Pro のチップ，16GB のメモリ，512GB のストレージを搭載している．M5Stack Fire はマイクロコントローラにディスプレイやボタンを組み合わせたモジュールである．内蔵されているマイクロコントローラは ESP32-D0WDQ6 で，520 KB の RAM，16 MB の Flash，8 MB の PSRAM を搭載している．ESP32-D0WDQ6 ではオーディオストリーミングなどに使用される Bluetooth Classic と超低消費電力の BLE (Bluetooth Low Energy) の二つのタイプの Bluetooth 通信が使用可能である．本実験では後者の BLE を使用した．また，ESP32-D0WDQ6 では 2.4 GHz で最大 150 Mbps で通信可能な Wi-Fi が搭載されている．M5Stack Fire に搭載されているディスプレイは 2inch の ILI9342C という機種で，SPI 通信経由でマイクロコントローラから操作することができる．M5Stack Fire にはボタンが 3 つ搭載されており，GPIO の電流の電圧の上下を読み取ることでボタンが押されたかどうかの判断ができる．

4.2　対話性

本実験では BlueScript の対話性が満たされていることを確認するために，BlueScript と MicroPython を使用した際にプログラムを実行しては改良，また実行，という手順を繰り返して最終的なプログラムを完成させるまでの過程の中で，プログラムの変更にかかる時間を計測した．

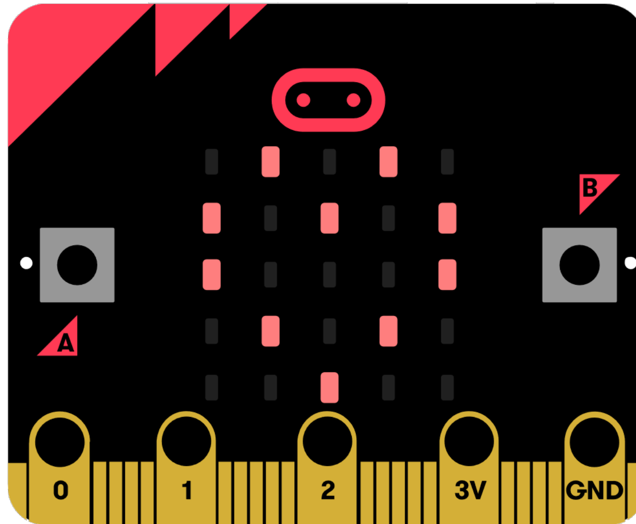

図 4.1. micro:bit

本研究では，2.3.1 節で述べたように言語処理系が対話性を備えているということを，次の二つの条件を満たしていることと同値であると考える．(1) 前の実行の結果を失わずにプログラムを追加・変更することができる．(2) 十分に早い時間でプログラムの追加・変更ができる．BlueScript で条件 1 が満たされていることは実装から明らかである．本実験では，BlueScript と MiroPython を使用して，BlueScript で条件 2 が満たされていることを確かめる．また，参考値として，C 言語でプログラムの変更にかかる時間も測定する．

4.2.1　ベンチマークプログラム

本実験では対話性の評価を行うために micro:bit[*1]のチュートリアルをベンチマークとして使用した．micro:bit とはイギリスで BBC Micro によって教育向けに開発されたマイコンボードであり，図 4.1 のように 25 個の LED とボタン，Bluetooth，加速度センサーを備えている．https://makecode.microbit.org/[*1]のウェブサイトからプログラムを書き，シミュレートして上手く動くことを確認した後，コンパイル済みのバイナリをシリアルケーブル経由で書き込むことで動作させることができる．micro:bit のチュートリアルとは，このウェブサイトで提供されている練習用のプログラム群である．ユーザは初心者用の簡単なプログラムを解説に沿って少しずつ追加，シミュレート機能で動く様子を見ることができる．使用可能な言語は Python，JavaScript，ブロックの三つである．このチュートリアルプログラムはオープンソースプロジェクトであり，Github のリポジトリ [1] から見ることができる．

プログラム 4.1 は micro:bit のチュートリアルの一つである，Flashing Heart の JavaScript プログラムである．このプログラムは四つの部分に分かれており，シミュレート機能で micro:bit の動きを見ながら各セルプログラムを追加していくことが想定されている．`basic`

[*1] https://makecode.microbit.org/

```
// Cell1
basic.showIcon(IconNames.Heart)

// Cell2
basic.showIcon(IconNames.Heart)
basic.clearScreen()
basic.pause(500)

// Cell3
basic.showIcon(IconNames.Heart)
basic.clearScreen()
basic.pause(500)
basic.showIcon(IconNames.SmallHeart)
basic.clearScreen()
basic.pause(500)

// Cell4
basic.forever(function() {
    basic.showIcon(IconNames.Heart)
    basic.clearScreen()
    basic.pause(500)
    basic.showIcon(IconNames.SmallHeart)
    basic.clearScreen()
    basic.pause(500)
})
```

Listing 4.1. micro:bit のチュートリアルの例

は micro:bit が提供しているライブラリである．チュートリアルは次のような手順で進める．(1) Cell1 のプログラムを入力する．シミュレート実行ボタンを押すと，LED 盤がハートの形に光る．(2) Cell2 のプログラムを入力する．シミュレート実行ボタンを押すと，LED 盤が一瞬ハートの形に光り，すぐに消える．(3) Cell3 のプログラムを入力する．シミュレート実行ボタンを押すと，LED 盤が一瞬ハートの形に光り，すぐに消えた後 500 ms 後に小さなハートの形に一瞬光る．(4) Cell4 のプログラムを入力する．シミュレート実行ボタンを押すと，500 ms 毎に普通の大きさのハートと小さいハートが一瞬光る動きを繰り返す．

本研究ではベンチマークプログラムとして micro:bit のチュートリアルから基本的な動きをする次のような 5 つのプログラムを抽出して使用した．

- Flashing Heart: 前述した例にあるプログラム．
- Name Tag: LED 盤に少しづつ文字を表示していき，最終的に”My name is: Sara! My age is: 9”と表示されるようにするプログラム．
- Smily Button: ボタンが押されると呼ばれる Callback 関数を書くところから始めて，

最終的にボタン A が押されたら笑った顔の顔文字を LED 盤に表示し，ボタン B が押されたら悲しい顔の顔文字を表示されるようにするプログラム．

- Dice: マイクロコントローラが振られたら呼ばれる Callback 関数を書くところから始めて，最終的にマイコンが振られたら 1 から 6 までの数字のうちどれかをランダムに選び LED 盤に表示するプログラム．
- Love Meter: タッチピンが押されたら呼ばれる Callback 関数を書くところから始めて，最終的にタッチピンが押されたら 0 から 100 までの数字のうちのどれかをランダムに選び LED 盤に表示するプログラム．

4.1 節で述べるように，実験には ESP32 が内蔵されている M5Stack というディスプレイやボタン，無線通信機能が搭載されているマイクロコントローラモジュールを使用した．本実験では，マイクロベンチマークをこの M5Stack に合う形に書き換えて使用した．具体的には，LED 盤をディスプレイに置き換え，タッチピンとシェイク感知機能をボタンに置き換えた．また，それらを使用するためのライブラリは C 言語，BlueScript，MicroPython でそれぞれの言語に合う形で実装した．C 言語では ESP-IDF が提供している SPI ライブラリと GPIO ライブラリを使用して C 言語でライブラリを実装した．BlueScript ではさらに C 言語で実装したライブラリを BlueScript で使用できるようにインターフェースを実装した．MicroPython では，MicroPython が提供している `machine` ライブラリの SPI 及び GPIO を扱うためのクラスを使用して MicroPython でライブラリを実装した．

4.2.2 測定方法

本研究では，BlueScript，MicroPython の対話性を評価した．具体的には，計測した時刻及び時間からコンパイル時間，通信にかかった時間，実行時間を算出した．また，比較のために C 言語で開発した場合のプログラムの変更にかかる時間も測定した．

図 4.2 は BlueScript，MicroPython，C 言語で開発した場合の実行ボタンを押してから，結果を受け取るまでの間に，マイクロコントローラとホストマシンで行われる動作を表した模式図である．薄いオレンジ色の矢印上に書かれた動作がホストマシン上で行われる動作であり，薄い青色の矢印上に書かれた動作がマイクロコントローラ上で行われる動作である．BlueScript を例に取ると，コンパイルがホストマシン上で行われ，実行がマイクロコントローラ上で行われる．また，濃いオレンジ色で指されている部分がホストマシン上で計測した時刻であり，青い矢印で示されている部分がマイクロコントローラ上で計測した時間である．本節ではそれぞれの言語でこれらの時刻と時間を計測した方法とそこからコンパイル時間，通信にかかった時間，実行時間を算出した方法について述べる．

BlueScript

実験には 4.1 節で記述されているホストマシンとマイクロコントローラ，及び Bluetooth を使用した．各セルのプログラムの最後にログを出力する式を挿入し，そのログが返された時刻

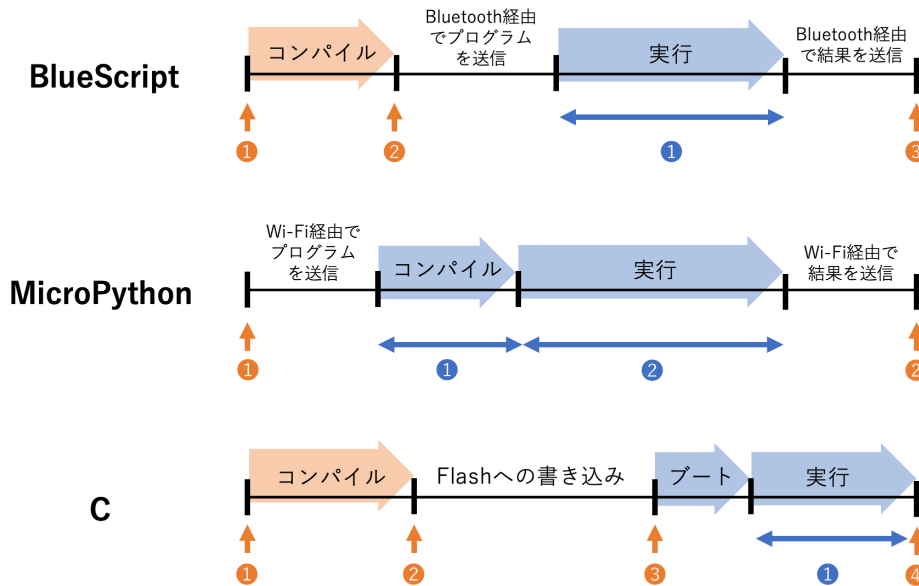

図 4.2. 対話性の評価で計測した時刻と時間

を結果が返された時刻とした．

ベンチマークの取り方は次のような手順で行った．

1. BlueScript のランタイムをマイクロコントローラにシリアルケーブル経由で書き込む．
2. ホストマシン上でサーバを立ち上げ，BlueScript のコードを書くためのブラウザのページを開く．
3. ブラウザのページの機能を使用してマイコンとホストマシンを Bluetooth で接続する．
4. セルにプログラムを入力して実行ボタンを押す．

図 4.2 の BlueScript の部分に濃いオレンジ色の矢印 1 から 3 で指されている時刻はホストマシンのブラウザ上で計測した時刻である．全てブラウザ上で `performance.now()` メソッドを使用して計測した．矢印 1 の時刻はブラウザ上で最新のプログラムが書かれたセルの実行ボタンが押された時刻である．矢印 2 の時刻はホストマシン上のサーバからコンパイル済みの機械語が返された時刻である．矢印 3 の時刻はホストマシン上でマイクロコントローラからログが返ってきた時刻である．

図 4.2 の BlueScript の部分に濃い青色で示されている部分がマイクロコントローラ上で計測した時間である．本実験では新しく送られてきたプログラムの実行する時間を測るためのプログラムを BlueScript ランタイムに挿入することで計測した．

これらの時刻と時間からコンパイル時間，通信にかかった時間，実行時間を算出した．コンパイル時間は濃いオレンジ色の矢印 1 と濃いオレンジ色の矢印 2 の差を取ることで算出した．通信にかかった時間は濃いオレンジ色の矢印 3 の時刻と濃いオレンジ色の矢印 2 の時刻の差

```
1 dspl = Display();dspl.show_icon(dspl.ICON_HEART, color_red, color_white);
     print("$$")
```

Listing 4.2. MicroPython で Flashing Heart のセル 1 のプログラムを書く例

をとり，そこから濃い青色の矢印 1 の時間を引くことで算出した．実行時間は濃い青色の矢印で示された時間をそのまま実行時間とした．

MicroPython

MicroPython の対話性を評価するために，MicroPython の WebREPL の画面からプログラムを入力して実行する場合の時刻と時間を計測した．実験には 4.1 節で記述されているホストマシンとマイクロコントローラ，及び Wi-Fi を使用した．

ベンチマークの取り方は以下のような手順で行った．

1. MicroPython のランタイムをマイクロコントローラにシリアルケーブル経由で書き込む．
2. シリアルケーブル経由で Wi-Fi の設定をする．
3. シリアルケーブル経由で自作の MicroPython ライブラリを送信する．
4. WebREPL の画面を開く．
5. WebREPL の画面からプログラムを書いて `Enter` キーを押しプログラムを実行することを繰り返す．

MicroPython の WebREPL の画面では，1 回の実行で 1 文しか実行することができない．そのため，4.2 のような形で式を; で区切ってつなげることで実行した．また，関数定義を含む場合など，一文に繋げられない場合は複数セルに分けた．また，結果を受け取った時刻を計測するために各文の最後に `print` 関数を呼び出す式を挿入した．

図 4.2 の MicroPython の部分に濃いオレンジ色の矢印 1，2 で示されている時刻がホストマシンのブラウザ上で計測した時刻である．全てブラウザ上で `performance.now()` メソッドを MicroPython が提供する WebREPL のソースコードに挿入することで計測した．矢印 1 の時刻はブラウザ上で新しいプログラムを入力した後にプログラムを実行するために `Enter` キーを押した時刻である．矢印 2 の時刻はブラウザ上でマイクロコントローラから `print` 結果を受け取った時刻である．関数を定義する文の最後には `print` 関数を呼び出す式を挿入することができなかったため矢印 2 の時刻を計測することができなかった．

図 4.2 の MicroPython の部分に濃い青色で示されている部分 1 と 2 がマイクロコントローラ上で計測した時間である．これらの時刻は MicroPython のランタイムのプログラムに計測を行うためのプログラムを挿入することで計測した．矢印 1 で示されている時間は MicroPython のランタイムが新しく送られてきたプログラムをバイトコードにコンパイルするのにかかった時間である．矢印 2 で示されている時間は MicroPython のランタイムがバイトコードを実行するのにかかった時間である．ベンチマークプログラムのうち，Flashing

```
1 (echo "helloworld" && idf.py build flash monitor )| ts "[%.s]"
```

Listing 4.3. C 言語で時刻を計測するために使用したコマンド

Heart のように無限ループがあるプログラムでは新しくスレッドを作成し，そのスレッドを作成するのにかかった時間を実行時間とした．

これらの時刻と時間からコンパイル時間，通信にかかった時間，実行時間を算出した．コンパイル時間は濃い青色の矢印 1 の時間をそのままコンパイル時間とした．通信にかかった時間は濃いオレンジ色の矢印 2 の時刻と濃いオレンジ色の矢印 1 の時刻の差をとり，そこから濃い青色の矢印 1 と 2 の時間を引くことで算出した．実行にかかった時間は濃い青色の矢印 2 の時間をそのまま実行時間とした．関数定義を別のセルに分離した場合の関数セルの実行時間は空欄とした．

C 言語

C 言語でのプログラムの変更にかかる時間を計測する実験では，4.1 節で記述されているホストマシンとマイクロコントローラを使用した．

ベンチマークの取り方は以下のような手順で行った．

1. 空の関数を実行するプログラムを書き，コンパイルとフラッシュへの書き込み，実行を行う．
2. ベンチマークプログラムの 2 番目のセルのプログラムを実行する関数とそれを呼び出すプログラム書き，コンパイルとフラッシュへの書き込み，実行を行う．
3. ベンチマークプログラムの 2 番目のセルのプログラムを 1 番目のプログラムの下に追加し，コンパイルとフラッシュへの書き込み，実行を行う．
4. 3 番目，4 番目のセルのプログラムについても同様に追加する．

手順の最初で空の関数を実行するプログラムのコンパイルとフラッシュへの書き込み・実行を行った理由は，キャッシュを作成するためである．ESP32 のコンパイルツールには毎回全てのプログラムを最初からコンパイルし直すとコンパイルに時間がかかり過ぎてしまうため，前回のコンパイル結果をキャッシュに残しておき，変更箇所に関わる部分のみをコンパイルすることでコンパイル時間を短くする仕組みがある．その仕組みを利用するため，1 番目のセルをコンパイルする前に空の関数を実行するプログラムをコンパイルした．

結果が表示される時刻を計測するために，ベンチマークプログラムを呼び出す関数の実行の直後に `printf` 関数を呼び出す文を挿入した．

図 4.2 の C 言語の部分に濃いオレンジ色の矢印 1 から 4 で示されている部分がホストマシン上で計測した時刻である．これらの時刻はプログラム 4.3 に書かれているコマンドを使用して計測した．`ts` コマンドはタイムスタンプを入力の各行に追加するコマンドである．このコマンドによって，各行がコマンドライン上に表示された時刻を計測した．`echo "helloworld"`はコマンドライン上に `helloworld` の文字を表示するコマンドである．このコ

表 4.1. BlueScript と MicroPython でのコンパイル・通信・実行時間の平均

言語	コンパイル時間 (ms)	通信時間 (ms)	実行時間 (ms)
BlueScript	236.1	291	0.148
MicroPython	4.745	204	2007

表 4.2. 各ベンチマークにおける BlueScript と MicroPython のコンパイル時間の平均

ベンチマーク	BlueScript (ms)	MicroPython (ms)
Dice	248.5	4.900
Flashing Heart	241.6	4.646
Love Meter	230.9	5.739
Name Tag	234.6	3.982
Smily Buttons	221.2	4.690

マンドが表示された時刻を図 4.2 の C 言語の部分の濃いオレンジ色の矢印 1 の時刻とした．`idf.py build flash monitor` は ESP-IDF で提供されているコマンドで，このコマンドが実行されたフォルダ内にある C 言語のプログラムをコンパイルしフラッシュに書き込んだ後，実行を開始してその実行結果をコマンドライン上に表示する．このコマンドを実行した後，コンパイルが完了したことを示す文字がコマンドライン上に表示された時刻を濃いオレンジの矢印 2 の時刻とした．また，フラッシュへの書き込みが完了したことを示す文字がコマンドライン上に表示された時刻を濃いオレンジの矢印 3 の時刻とし，各セルのプログラムの最後に追加した `print` の結果がコマンドライン上に表示された時刻を濃いオレンジの矢印 4 の時刻とした．

図 4.2 の C 言語の部分に濃い青色で示されている部分 1 がマイクロコントローラ上で計測した時間である．この時間はベンチマークプログラムを実行する関数の前後に時間を測るためのプログラムを挿入することで計測した．ベンチマークプログラムのうち，Flashing Heart のように無限ループがあるプログラムでは新しくスレッドを作成し，そのスレッドを作成するのにかかった時間を実行時間とした．

これらの時刻と時間からコンパイルとフラッシュへの書き込み及び実行にかかった時間を計測した．コンパイルにかかった時間は濃いオレンジの矢印 1 と 2 の差を取ることで算出した．Flash に書き込むのにかかった時間は濃いオレンジの矢印 2 と 3 の時刻の差を取ることで算出した．実行にかかった時間は濃い青矢印が示す時間をそのまま実行時間とした．

4.2.3 結果

表 4.1 は BlueScript と MicroPython でのコンパイル時間・通信にかかった時間・実行時間の平均を比較した表である．また，表 4.2 と表 4.3 は各ベンチマークにおける BlueScript と MicroPython でのコンパイル時間と送信時間の平均を示したものである．各ベンチマークプログラムについて，それぞれ 3 回分のデータをとり，外れ値を抜いて全ての平均をとった．例えば，セルが 3 つあるベンチマークプログラムでは 9 回分のコンパイル時間・通信にかかった

表 4.3. 各ベンチマークにおける BlueScript と MicroPython の送信時間の平均

ベンチマーク	BlueScript (ms)	MicroPython (ms)
Dice	311	192
Flashing Heart	338	257
Love Meter	375	263
Name Tag	230	176
Smily Buttons	219	166

表 4.4. C 言語でのコンパイル・フラッシュへの書き込み・実行時間の平均

言語	コンパイル時間 (s)	フラッシュへの書き込み時間 (s)	実行時間 (s)
C 言語	4.6655	3.369	0.583

時間・実行時間を測り平均をとった．コンパイル時間・通信にかかった時間・実行時間の測定方法については 4.2.2 で述べた通りである．

表 4.1 より，BlueScript と MicroPython の時間を比較するとコンパイル時間についてはBlueScript の方が MicroPython よりも平均約 230ms 遅いことがわかる．この差は，BlueScript ではローカルサーバとの通信が必要なことと，GCC を用いてより最適化された機械語を生成しているためであると考えられる．通信時間については BlueScript の方が MicroPython に比べて平均約 90 ms 遅くなっている．この差は BlueScript が Bluetooth を用いて通信を行っているのに対し，MicroPython が Wi-Fi で通信を行っているためではないかと思われる．また，実行時間は MicroPython の方が BlueScript よりも大幅に遅くなっているが，これは MicroPython の Display ライブラリは MicroPython で実装したのに対し，BlueScript は C で実装してインターフェースだけ提供したからである．実行時間を抜いた，プログラムの変更にかかる時間を比べると，BlueScript での時間は MicroPython の時間の約 2.5 倍程度であることがわかる．このことから，BlueScript の対話性は MicroPython の対話性とほぼ同等であることがわかる．

表 4.4 は C 言語でコンパイル・フラッシュへの書き込み・ユーザプログラムの実行にかかった時間の平均を示した表である．表 4.1 ではミリ秒の単位を使用していたが，表 4.4 では見やすいように単位を秒に変更した．この表から C 言語ではコンパイルとフラッシュへの書き込みに平均 8.37 秒かかっていることがわかる．これは BlueScrit でコンパイルと通信にかかる時間に比べて非常に遅く，プログラムの変更の反映が煩雑であることがわかる．

4.3 実行速度

本実験ではいくつかのベンチマークプログラムを使用して，BlueScript・MicroPython・C 言語でのプログラムの実行速度を比較した．

4.3.1 ベンチマーク

ベンチマークプログラムとして Marr らの Cross language compiler benchmarking[2] と ProgLangComp[3] のマイクロベンチマークを C 言語， MicroPython， BlueScript で書き直したものを使用した．AreWeFastYet は言語実装を比較するために作成されたベンチマークである．本実験ではその中でも以下のようなマイクロベンチマークに分類されるものを使用した．

- Bounce：箱の中でボールがバウンドする様子をシミュレートする．
- List：再帰的にリストを作成.
- Msndelbrot：古典的なフラクタルを作成.
- Nbody：太陽系の惑星の動きをシミュレートする．
- Permute：配列の順列を作成.
- Queens：エイト・クイーン問題を解く．
- Sieve：エラトステネスのふるいに基づいて素数を発見.
- Storage：配列のツリーを作成．ガベージコレクタに負荷をかける．
- Towers：ハノイの塔の問題を解く．

ProgLangComp はマイクロコントローラ上で言語実装を比較するために作成されたベンチマークであり，マイクロコントローラ上で行われれることが多い信号処理などのアルゴリズムが多く含まれている．ベンチマークプログラムは以下のようなものがある．

- Biquad：双 2 次フィルタ
- CRC：巡回冗長検査
- SHA256：ハッシュ生成方法
- FFT：高速フーリエ変換
- IIR：ディジタルフィルタの一種
- FIR：ディジタルフィルタの一種

4.3.2 測定法

各ベンチマークプログラムにつき，タイマーの誤差を減らすために 3 回実行する時間を測り，3 で割ることを 5 回繰り返してその平均をとった．BlueScript では，BlueScript で書かれたプログラムを C 言語への変換器にかけて変換し，BlueScript ランタイムと一緒にコンパイル・実行した．MicroPython ランタイムをマイクロコントローラに書き込んだ後，ampy コマンドを使って MicroPython プログラムをマイクロコントローラに送り，実行した．C 言語では，ESP-IDF のツールを使用してコンパイル・実行した．

表 4.5. BlueScript，MicroPython，C 言語における実行時間

ベンチマーク名	BlueScript(ms)	MicroPython(ms)	C(ms)	BlueScript/C	MicroPython/C
Sieve	3.16	92.31	1.09	2.89	84.37
Nbody	18991.39	472416.52	17300.30	1.10	27.31
Permute	11.52	342.34	2.14	5.39	160.27
Storage(5)	1.08	16.42	7.55	0.14	2.17
Storage(6)	4.76	67.03	30.22	0.16	2.22
Queens	5.70	233.25	1.32	4.32	176.84
Towers	25.22	574.84	3.38	7.47	170.24
List	13.69	170.01	1.54	8.91	110.69
Bounce	3.89	205.63	1.57	2.48	131.31
Mandelbrot	191.59	null	63.70	3.01	null
Biquad	5.80	111.73	2.10	2.76	53.18
FIR	96.97	5971.72	17.66	5.49	338.21
CRC	0.90	107.48	0.18	5.05	604.96
FFT	10.99	328.09	0.94	11.73	350.40
SHA256	3.67	1140.26	0.93	3.96	1232.72

4.3.3　結果

表 4.5 は各ベンチマークプログラムの BlueScript，MicroPython，C 言語での実行時間を比較したものである．Storage(5) と Storage(6) はそれぞれ Storage ベンチマークプログラムでツリーの深さを 5 と 6 にしたものである．また，Mandelbrot ベンチマークプログラムでは MicroPython の場合に精度が少し落ちて，プログラム中のループの数を合わせることができなかったため `null` とした．

表 4.5 から，BlueScript での実行時間が MicroPython での実行時間に比べて大幅に短いことがわかる．BlueScript での実行時間は C 言語での実行時間の 1.1 倍から 11.7 倍に収まっていたのに対し，MicroPython での実行時間は C 言語での実行時間の 2.1 倍から 1232 倍であった．特に，ProgLangComp[3] の信号処理のベンチマークプログラムで BlueScript と MicroPython の差が大きく出ている．この差は，BlueScript の場合 `integer` 配列や `float` 配列では要素の型検査をやらずに済むのに対して，MicroPython では毎回要素の型検査が必要であるためではないかと考えられる．

BlueScript と C 言語の実行時間を比べると，FFT ベンチマークプログラムで BlueScript での実行が遅くなっている．これは，オブジェクトのプロパティへのアクセスが頻繁に起こったためではないかと考えられる．また，Towers と List のベンチマークプログラムで比較的 BlueScript での実行が遅くなっている．これは，関数の呼び出しが頻繁に行われているため関数の間接参照のオーバーヘッドが出たのではないかと予想される．一方で Storage ベンチマークプログラムでは C 言語よりも BlueScript の方が実行時間が短くなっている．これは C 言語での場合はメモリを効率的に使用するためのアロケータを使用しているのに対し，BlueScript

ではデフォルトのアロケータを使用せず，最初に大きなメモリプールを確保した後単純なフリーリスト方式でメモリを管理しているためであると考えられる．

4.4 バイナリサイズ

表 4.6. BlueScript と MicroPython のランタイムのバイナリサイズ

言語	バイナリサイズ (MB)
BlueScript	0.64
MicroPython	1.67

本実験では，BlueScript と MicroPython のランタイムのバイナリサイズを比較した．バイナリサイズは `esptool.py image_info` コマンドを使用して，各言語の ESP32 向けにコンパイルされたイメージの大きさを調べた．

表 4.6 を見るとわかる通り，MicroPython が 1.67 MB であるのに対して，BlueScript のランタイムのバイナリの大きさが 0.64 MB であった．このことから，動的コンパイラを使用しているにも拘らず，BlueScript でのバイナリサイズが小さいことがわかる．しかし，MicroPython では BlueScript に比べて標準ライブラリなどは充実しているため，この実験についてはさらなる検証が必要である．

第 5 章

まとめと今後の課題

本論文では，対話的でありながら実行速度の速いマイクロコントローラ向けの開発環境を実現するために，動的コンパイラのホストマシンへのオフロードを提案した．対話的に入力されたプログラム片をマイクロコントローラに比べて計算リソースの多いホストマシン上で，機械語にコンパイルすれば，マイクロコントローラのメモリを圧迫することなく十分な最適化を施すことができる．本研究では提案手法を BlueScript の処理系として実装した．BlueScript は対話的な開発環境を持つマイクロコントローラ向けの TypeScript 風言語である．評価として BlueScript，MicroPython でプログラムの変更にかかる時間を測定し，BlueScript が MicroPython と同程度の対話性を持つことを示した．さらに，BlueScript，MicroPython，C 言語でベンチマークの実行時間を比較し，MicroPython での実行時間が C 言語での実行時間に比べて 2.1 倍から 1232 倍であったのに対し，BlueScript では C 言語の 1.1 倍から 11.7 倍であることを示した．

発表文献と研究活動

(1) 前島文香. 山崎 徹郎. 千葉 滋. 対話性と十分な実行速度を両立した組み込みマイコン向け開発環境の提案. 日本ソフトウェア科学会 第 40 回大会. 2023.09.12-14.

参考文献

[1] Microsoft Corporation. Pxt - programming experience toolkit. `https://github.com/microsoft/pxt-microbit/`.

[2] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: Are we fast yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, p. 120–131. Association for Computing Machinery, 2016.

[3] Ignas Plauska, Agnius Liutkevičius, and Audronė Janavičiūtė. Performance evaluation of c/c++, micropython, rust and tinygo programming languages on esp32 microcontroller. *Electronics*, Vol. 12, No. 1, 2023.

[4] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, Vol. 35, No. 2, p. 97–113, jun 2003.

[5] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, p. 1–12, New York, NY, USA, 2000. Association for Computing Machinery.

[6] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, p. 89–100, New York, NY, USA, 2007. Association for Computing Machinery.

[7] Andreas Gal, Christian W. Probst, and Michael Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, p. 144–153, New York, NY, USA, 2006. Association for Computing Machinery.

[8] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. Adaptive multi-level compilation in a trace-based java jit compiler. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, p. 179–194, New York, NY, USA, 2012. Association for Computing Machinery.

[9] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM*

Systems Journal, Vol. 39, No. 1, pp. 175–193, 2000.

[10] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, Vol. 44, No. 6, p. 465–478, jun 2009.

[11] V8. `https://v8.dev/`, 2014.

[12] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, p. 18–25, New York, NY, USA, 2009. Association for Computing Machinery.

[13] Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. Yjit: A basic block versioning jit compiler for cruby. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2021, p. 25–32, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Geetha Manjunath and Venkatesh Krishnan. A small hybrid jit for embedded systems. *SIGPLAN Not.*, Vol. 35, No. 4, p. 44–50, apr 2000.

[15] Giuseppe Di Giore, Antonella Di Stefano, Giovanni Morana, and Corrado Santoro. Jit compiler optimizations for stack-based processors in embedded platforms. In *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '06, p. 212–217, New York, NY, USA, 2006. Association for Computing Machinery.

[16] Yuan Zhang, Min Yang, Bo Zhou, Zhemin Yang, Weihua Zhang, and Binyu Zang. Swift: a register-based jit compiler for embedded jvms. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, p. 63–74, New York, NY, USA, 2012. Association for Computing Machinery.

[17] Henri-Pierre Charles and Victor Lomüller. Is dynamic compilation possible for embedded systems? In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, p. 80–83, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Joel Koshy, Raju Pandey, and Ingwar Wirjawan. Optimizing embedded virtual machines. In *2009 International Conference on Computational Science and Engineering*, Vol. 2, pp. 342–351, 2009.

[19] Thomas Ball, Peli de Halleux, and Michał Moskal. Static typescript: An implementation of a static compiler for the typescript language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and*

Runtimes, MPLR 2019, p. 105–116. Association for Computing Machinery, 2019.

[20] Jeremy Siek and Walid Taha. Gradual typing for functional languages. 01 2006.

[21] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[22] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly darbage collection: an exercise in cooeration. In *Language Hierarchies and Interfaces*, pp. 43–56, Berlin, Heidelberg, 1976. Springer Berlin Heidelberg.

謝辞

本論文は，東京大学情報理工学系研究科創造情報学専攻千葉研究室において行った研究の成果をまとめたものである．

研究の遂行にあたり，お忙しい中でも毎週ミーティングの時間を設け，丁寧にご指導くださった千葉教授に心から感謝いたします．研究の方針や発表スライドの作り方，論文の書き方など大変多くのことを教えていただきました．また，先生が書いて下さったソースコードを読むことで多くの知見を得ることができました．日頃より親身になってご相談に乗って頂いた山崎特任教授に深く感謝いたします．プログラミング言語に関する知識や論文の書き方，スライドを作る際の考え方など多くの面でご指導いただきました．最後に毎週のミーティングでご指導いただいた鵜川准教授，研究を研究を進める上でご相談させていただきました先輩方，論文を書く間励ましあった同期に深く御礼申し上げます．本当にありがとうございました．